Université de Pau et des Pays de l'Adour
UFR Sciences et Techniques
Département informatique

Master Technologies de l'Internet
2ème année

Rapport de stage recherche

Architecture logicielle pour capteurs sans-fil en réseau

Par :
Séverine Sentilles

Encadrants : Nicolas Belloir
CongDuc Pham

Janvier - juin 2006

Table des matières

I. Introduction ---- **- 1 -**

II. Etat de l'art ---- **- 3 -**

II.1) Les réseaux de capteurs ---- *- 3 -*
II.1.1) Les capteurs « traditionnels » ---- - 3 -
II.1.2) Les capteurs dans les Wireless Sensor Networks ---- - 4 -
II.1.3) La mise en réseau ---- - 7 -
II.1.4) Les différentes utilisations des réseaux WSNs ---- - 8 -
II.1.5) Les caractéristiques des réseaux de capteurs sans-fil ---- - 8 -
II.1.6) Les challenges/Les besoins ---- - 10 -

II.2) TinyOS et sa distribution ---- *- 11 -*
II.2.1) Les notions principales ---- - 11 -
II.2.2) NesC ---- - 14 -
II.2.3) La compilation ---- - 17 -

II.3) Etat de l'art sur la CBSE ---- *- 17 -*
II.3.1) Les concepts fondamentaux de la CBSE ---- - 17 -
II.3.2) Le modèle initial de Think ---- - 19 -
II.3.3) Fractal ---- - 23 -
II.3.4) Le modèle Think-Fractal ---- - 26 -

III. Vers une architecture logicielle pour les réseaux de capteurs ---- **- 27 -**

III.1) Les limites de TinyOS ---- *- 27 -*

III.2) Vers un système d'exploitation basé composants ---- *- 28 -*

IV. Etude de la reconfiguration dynamique ---- **- 35 -**

IV.1) Présentation de la solution proposée par Juraj Polakovic ---- *- 35 -*

IV.2) Les améliorations du modèle ---- *- 37 -*

V. Les limites de Think pour implémenter le modèle proposé ---- **- 41 -**

VI. Conclusion ---- **- 42 -**

Bibliographie ---- **- 43 -**

Table des figures ---- **- 46 -**

Annexes ---- **- 47 -**

Annexe A : Les composants systèmes fournis pour PowerPC dans la bibliothèque Kortex ---- *- 48 -*

Annexe B : Manuel d'installation de Think-v2 pour Ubuntu ---- *- 49 -*

Annexe C : Manuel d'installation de Think-v3 pour Ubuntu ---- *- 52 -*

Annexe D : Manuel d'installation et d'utilisation de l'émulateur ARM SkyEye-v1 (pour Ubuntu) ---- *- 55 -*

Annexe E : Développement d'une application de test ---- *- 56 -*

Annexe F : Manuel de l'IDL ---- *- 60 -*

I. Introduction

Les avancées technologiques récentes confortent la présence de l'informatique et de l'électronique au coeur du monde réel. De plus en plus d'objets se voient ainsi équiper de processeurs et de moyens de communication mobiles leur permettant de traiter des informations mais également de les transmettre. Cette évolution s'inscrit dans le cadre de *l'informatique pervasive*, plus connue sous le nom *d'ubiquitous computing* [Wei96]. Un des objectifs de ce domaine est de combler le fossé entre les mondes réel et virtuel en rendant les objets « intelligents ». Pour cela, ceux-ci doivent être capables de détecter un changement dans leur environnement et d'y réagir en fonction notamment des besoins de l'utilisateur.

Les réseaux de capteurs sans-fil entrent dans ce cadre. En effet, ceux-ci sont constitués d'un ensemble de petits appareils, ou capteurs, possédant des ressources particulièrement limitées mais qui leur permettent néanmoins d'acquérir des données sur leur environnement immédiat, de les traiter et de les communiquer. Toute une gamme de nouvelles applications peut dès lors être envisagée.

Cependant, l'intégration de ces capteurs dans le monde physique n'est pas une tâche aisée. En effet, de nouveaux problèmes apparaissent engendrés entre autre par la sévérité des contraintes inhérentes aux ressources limitées. Citons notamment la conservation de l'espace mémoire ou encore la gestion limitée de l'énergie. Afin de répondre à ces exigences, TinyOS, le système d'exploitation de référence, occupe très peu de mémoire grâce aux nombreuses optimisations élaborées à partir de l'utilisation d'une architecture basée composant.

D'un autre côté, l'ingénierie logicielle basée composant[1] [SGM02] est une approche maintenant reconnue permettant le développement d'applications et de systèmes modulables, flexibles et bien architecturés, répondant notamment à des besoins de reconfiguration et d'administration. Or même si TinyOS présente des concepts similaires à ceux présents dans l'ingénierie logicielle basée composant, tous ne sont pas repris. En particulier, aucun support n'est proposé pour la gestion des aspects dynamiques dont une application pourrait avoir besoin. Par exemple, il est impossible de reconfigurer dynamiquement une application. Or différents travaux notamment sur l'*autonomic computing* [KC03] ont identifié cette fonctionnalité comme primordiale. L'utilisation de modèles réellement basés composant semble donc être une approche à approfondir. Dans ce cadre, il existe un modèle permettant de créer n'importe quel type de système d'exploitation. Il s'agit du modèle Think [Th06], une implémentation de Fractal [BCS04], qui cible les systèmes fortement contraints notamment les systèmes embarqués.

C'est dans ce contexte que s'inscrivent une partie des recherches du projet « Architectures Logicielles, Composant et protocoL (ALCooL) » du LIUPPA[2] au sein duquel ce stage a été réalisé. Notre objectif est d'expliquer l'apport de l'ingénierie logicielle basée composant dans le domaine des réseaux de capteurs sans-fil. Plus particulièrement, nous avons étudié la possibilité d'utiliser une plate-forme à composant telle que Think dans les réseaux de capteurs et de proposer une architecture logicielle qui permette des services avancés telle que la reconfiguration dynamique. Pour répondre à cette problématique, la démarche suivante a été suivie :

[1] En anglais *Component-Based Software Engineering* (CBSE)
[2] Laboratoire Informatique de l'Université de Pau et des Pays de l'Adour

- Etudier le système d'exploitation TinyOS afin d'en comprendre le fonctionnement et d'en établir les limites ;
- Explorer le modèle Think dans l'optique de proposer les bases d'un modèle pour un nouveau système d'exploitation mariant les fonctionnalités éprouvées de TinyOS et celles de l'ingénierie logicielle basée composant ;
- Proposer un modèle de mécanisme de reconfiguration dynamique adapté au domaine des réseaux de capteurs.

Dans ce but, nous allons dans un premier temps présenter à la section II les connaissances nécessaires au cadre de ce travail en décrivant les réseaux de capteurs, TinyOS, l'ingénierie logicielle basée composant et Think . Dans un deuxième temps, nous détaillerons à la section III les fondements de la création d'un nouveau système d'exploitation basé composant pour les réseaux de capteurs. Puis dans un troisième temps, nous exposerons à la section IV un mécanisme de reconfiguration dynamique s'appuyant sur le système d'exploitation précédemment défini. Pour terminer, nous établirons les limitations pratiques de notre travail dans la section IIV.

II. Etat de l'art

Cette section présente les notions fondamentales servant de socle à cette étude. Ainsi, nous détaillerons d'abord le domaine des réseaux de capteurs, puis TinyOS le système d'exploitation de référence utilisé. Et nous terminerons par une description de l'ingénierie logicielle basée composant ainsi qu'une présentation des modèles Think et Fractal.

II.1) Les réseaux de capteurs

Les réseaux de capteurs sans-fil, plus connus sous le nom de *Wireless Sensor Networks* (*WSNs*) sont une technologie émergente issue des progrès de différents domaines. Ils ont en effet profité de la miniaturisation des composants électroniques, de l'augmentation de leur capacité (puissance de calculs, énergie, etc…), de la diminution des coûts de fabrication mais également des avancées dans les réseaux de communication sans-fil à travers l'essor des téléphones mobiles, des PDAs, etc.

Cette section s'articule de la manière suivante. Nous présentons d'abord les capteurs « traditionnels » autrement dit les « ancêtres » de ceux présents dans les *WSNs* puis les nouveautés apportées par les *motes,* c'est-à-dire les capteurs utilisés dans les réseaux de capteurs sans-fil. Ensuite, nous décrivons l'aspect général d'un tel réseau en montrant les relations entre les différents éléments le constituant avant d'enchaîner avec les différentes utilisations possibles. Enfin, nous présentons les caractéristiques de tels réseaux avant de terminer avec les besoins et les challenges de ce domaine.

II.1.1) Les capteurs « traditionnels »

Les réseaux de capteurs sans-fil visent à étendre les domaines d'application des capteurs. En effet, des capteurs sont déjà utilisés pour la création de systèmes d'acquisition de données car ils sont capables de détecter des phénomènes dans un environnement proche, de les quantifier et de transmettre les données ainsi obtenues à d'autres éléments du système en vue de leur traitement. Les éléments mesurés sont des grandeurs physiques telles que la pression, l'humidité, les vibrations, etc. [Hub00]

Pour réaliser de telles mesures, ces capteurs sont constitués de différents éléments hardwares. Afin de les mettre en évidence, considérons par exemple, un système d'acquisition de données permettant de relever la température ambiante d'un milieu. Ce système est constitué d'un thermomètre électronique, lui-même composé d'un capteur et d'un afficheur digital. Une fois la température établie, la mesure est transformée en signal analogique grâce à l'élément de détection. Pour qu'elle puisse être affichée, il est nécessaire que ce dernier soit digitalisé. On utilise pour cela un convertisseur. De plus, la présence du dispositif électronique précédemment cité impose une source d'énergie qui peut être soit le secteur, soit des batteries, etc. Enfin, si le destinataire des données (dans notre exemple, l'afficheur) est éloigné du capteur, il faut que ce dernier puisse transmettre la donnée numérisée, il peut donc y avoir également un module de communication comme un réseau câblé par exemple.

Si l'on ne considère que les éléments impliqués dans l'exemple précédent alors le capteur ainsi schématisé serait incomplet. En effet, deux autres composants peuvent être intégrés dans un tel instrument : il s'agit du *conditionneur de signaux* et d'un *processeur de signal numérique*[3]. Le conditionneur de signaux intervient afin d'améliorer la qualité des mesures. En effet, sous certaines conditions, notamment environnementales, celles-ci peuvent être faussées c'est-à-dire que le signal émis peut présenter du bruit, que son amplitude peut-être trop faible, qu'il peut être biaisé ou dépendre de paramètre secondaire. Le processeur de signal numérique permet quant à lui de traiter le signal c'est-à-dire d'en extraire des données, de le modifier ou de l'adapter en vue de la transmission ou du stockage. De plus, l'élément de détection n'est pas toujours capable de mesurer la quantité intéressante mais une quantité reliée. Le conditionneur de signaux et le processeur de signal numérique permettent de résoudre ces problèmes respectivement par l'utilisation de filtre passe-bas, de mécanisme de compensation, d'amplificateur opérationnel, etc. [Lew04] et par des mécanismes de traitement du signal. La Figure 1 ci-dessous est une modélisation de ces différents éléments inspirés à la fois par [Hub00] et la représentation des capteurs proposée à la Figure 2.

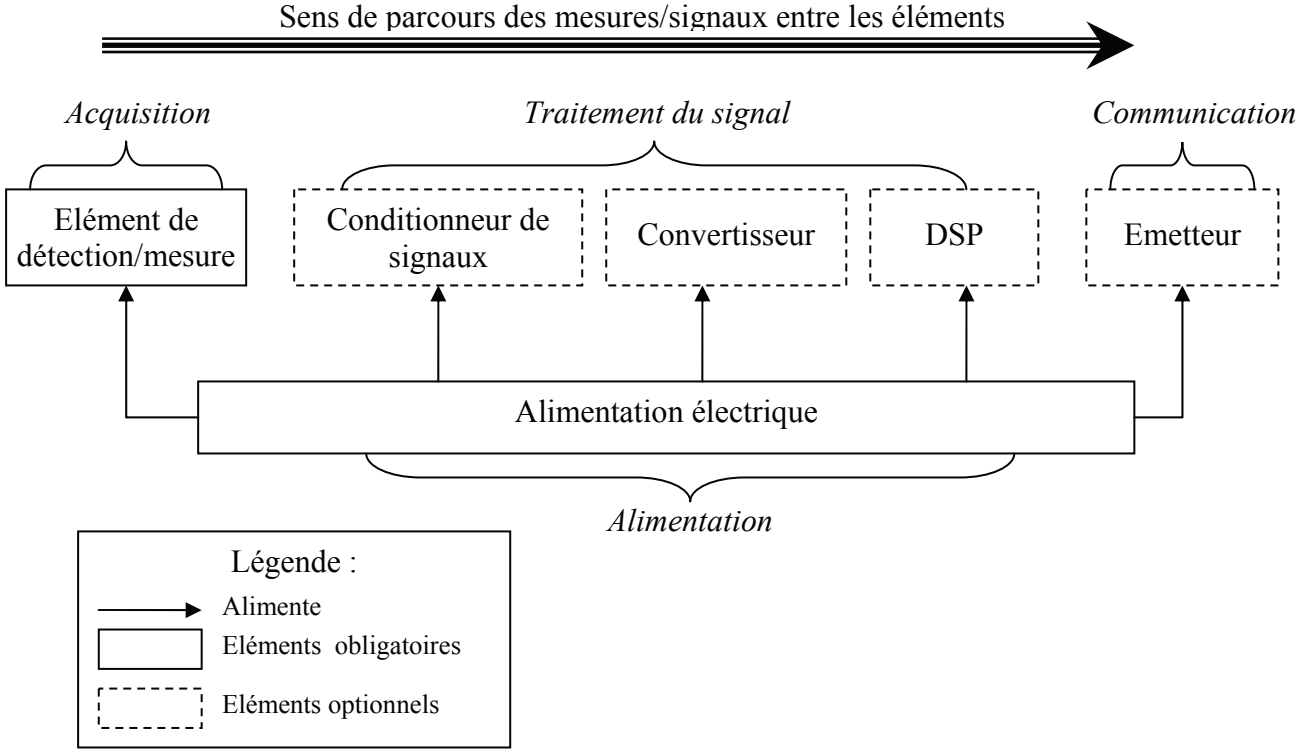


Figure 1 : Schématisation d'un capteur "traditionnel"

Après cette brève description des capteurs « traditionnels », nous en arrivons maintenant à la présentation des *motes* i.e. les capteurs utilisés dans les *WSNs*.

II.1.2) Les capteurs dans les Wireless Sensor Networks

Les capteurs présents dans les Wireless Sensor Networks se différencient de ceux précédemment décrits par plusieurs points. Le premier élément de distinction est révélé par le terme sans-fil. En effet, les *motes* ne sont plus câblés mais communiquent avec les autres éléments via un module de communication sans-fil.

[3] en anglais Digital Signal Processing (DSP)

De plus, alors que les capteurs précédents se contentaient d'envoyer des données à des éléments capables de les traiter, les *motes* peuvent également en recevoir. En effet, ils ont été conçus afin de fonctionner en réseau et afin de servir de relais pour que les données collectées et les autres informations essentielles au maintien du réseau puissent être propagées à travers celui-ci. Cette transmission est rendue possible grâce à un émetteur-récepteur[4]. Celui-ci communique selon trois modes : grâce à des lasers, des infrarouges ou des radiofréquences. Un des facteurs limitant dans l'utilisation du *transceiver* est la portée. En effet, il est à noter que la distance maximale de transmission entre deux nœuds du réseau est limitée à quelques dizaines de mètres. Un autre facteur limitatif est la consommation d'énergie du transceiver qui est relativement importante dans le cas d'une émission de données.

La seconde distinction est due non seulement à une diminution des coûts de production mais également à une miniaturisation de la taille des composants. Bien que ces progrès s'appliquent également aux capteurs décrits précédemment, il s'agit dans le cas des réseaux de capteurs d'une condition *sine qua non*. En effet, les capteurs ne sont désormais pas plus grands qu'une pièce de deux euros. Tous ces facteurs permettent d'envisager le déploiement d'un nombre beaucoup plus conséquent de senseurs[5] à savoir des centaines voire des milliers. Cela permet de créer un réseau beaucoup plus dense où la fiabilité de chaque nœud n'est plus primordiale. En effet, en imaginant un déploiement aléatoire, plusieurs nœuds pourraient se retrouver au même endroit et couvrir la même région. Ce qui permettrait à un nœud défaillant d'être remplacé par un autre.

La troisième différenciation se produit au niveau de la source d'énergie. En effet, dans le cas des *motes*, il s'agit d'une alimentation embarquée contrairement aux précédents où la source d'énergie pouvait être fournie par une source extérieure. Cela représente une contrainte importante pour la durée de vie des capteurs car la capacité des batteries est limitée et que, de plus, il n'est pas toujours possible de les remplacer lorsqu'elles sont déchargées. Cela se conçoit facilement si l'on envisage un très grand nombre de capteurs et/ou un environnement « hostile » de déploiement. Enfin, toute opération effectuée par les *motes* consomme de l'énergie entraînant ainsi une décharge plus ou moins importante des batteries. Parmi les opérations nécessitant énormément d'énergie, nous pouvons citer toutes celles impliquant l'émetteur-récepteur afin d'émettre des données [CES04]. Bien qu'il existe des batteries rechargeables par énergie solaire, par vibrations, etc., et des politiques de gestion de l'énergie, celles-ci ne suffisent pas, à elles seules, à permettre une utilisation prolongée des senseurs. L'économie d'énergie reste, par conséquent, un des points cruciaux dont il faut tenir compte lorsqu'on parle de *WSN*.

La dernière différence se situe entre la phase d'acquisition des données et la phase de communication. En effet, le principal avantage d'un *mote* est de disposer en plus d'une unité de traitement permettant non seulement de rapprocher le traitement des données de leur lieu de détection mais également de les agréger. Cela s'intègre dans le cadre des politiques d'économie d'énergie essentielles aux *WSNs*. En effet, cela permet de diminuer la taille des messages à transmettre aux autres éléments du réseau et ainsi de diminuer le temps d'utilisation de l'émetteur-récepteur qui, pour rappel, est l'élément consommant le plus d'énergie dans les *motes*. Pour pouvoir réaliser cette tâche, le capteur doit donc disposer également d'une unité de stockage nécessaire à l'implantation et à l'exécution d'un programme logiciel. Celle-ci est généralement de moins de 10ko de RAM et de moins de

[4] En anglais transceiver
[5] Autre terme pour désigner un capteur

100ko de ROM [CES04] et est donc de très faible capacité par rapport à ce que nous pouvons trouver sur les ordinateurs personnels actuels. Afin d'augmenter la taille de la mémoire disponible, il est également possible d'utiliser des mémoires additionnelles de type Flash ou Eprom par exemple. Un autre type de mémoire supplémentaire pourrait bientôt être envisagé : il s'agit de la *Magnetoresistive Random Access Memory* ou *MRAM*. En effet, celle-ci est en cours de développement mais ses atouts pour une utilisation dans les réseaux de capteurs pourraient être multiples car elle est très rapide à l'écriture et à la lecture, non volatile, elle consomme peu d'énergie, peut être effacée et réécrite un nombre de fois illimité (ce qui lui donne une espérance de vie aussi longue que celle de l'appareil qu'elle équipe), son coût de fabrication est faible et elle ne subit pas les influences des radiations [Mai06].

La Figure 2 présente les différents changements qui ont été apportés aux capteurs utilisés traditionnellement et que nous venons de décrire. Celle-ci est à mettre en parallèle avec la Figure 1 précédente.

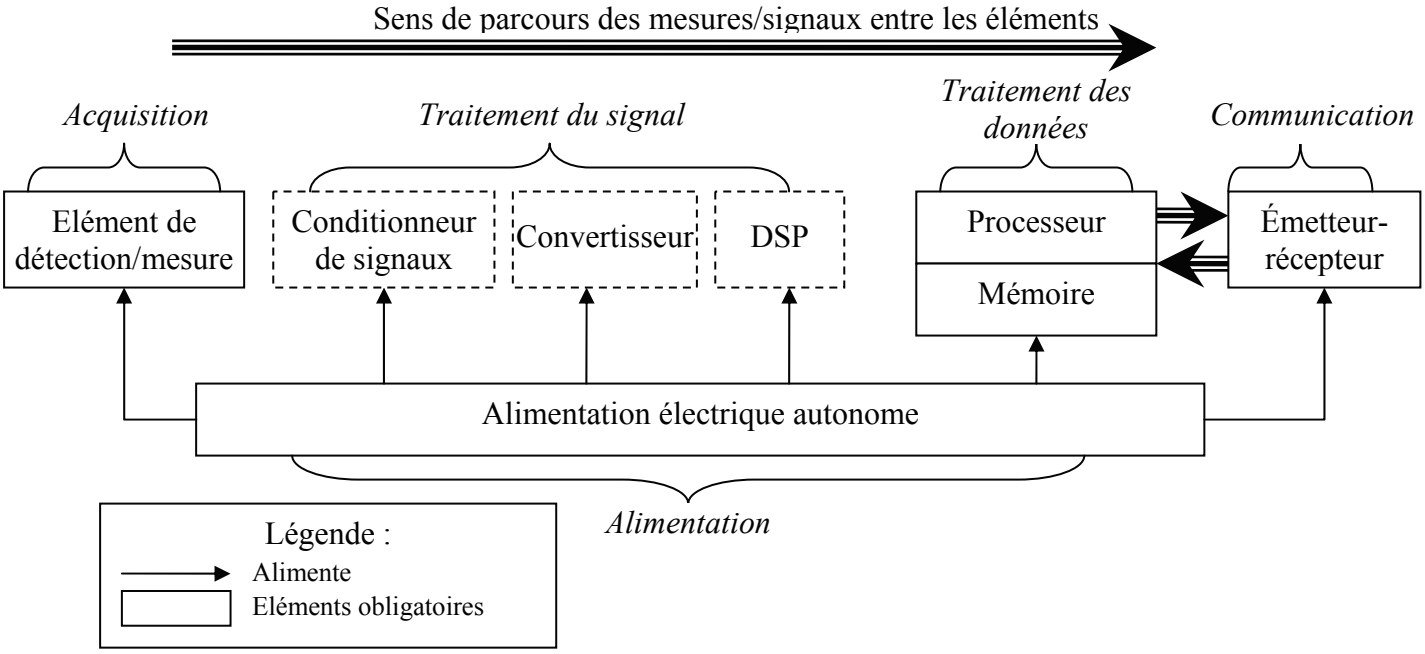


Figure 2 : Schéma d'un composant d'un réseau de capteurs, inspiré de [KDM05]

En outre, ces appareils peuvent être également appelés « capteurs intelligents » tel que la norme présentée dans [IEEE-1451 Expo2001] les définis. En effet, celle-ci standardise leurs caractéristiques à savoir la définition des interfaces pour qu'ils puissent se connecter à des réseaux divers, les fonctionnalités additionnelles qu'ils doivent posséder en plus de celles nécessaires à une représentation correcte du phénomène quantifié, etc. Parmi ces fonctions, nous pouvons citer : la facilité d'installation, l'auto-identification, l'auto-diagnostic, la fiabilité, le temps d'éveil pour la coordination avec d'autres nœuds, quelques fonctions logicielles, le traitement du signal, des protocoles de contrôles standards et des interfaces réseaux [IEEE-1451 Expo2001]. De plus, les motes tout comme cette norme visent les mêmes objectifs à savoir rapprocher l'intelligence du point de mesure et minimiser leur coût d'intégration ou de maintenance dans des réseaux distribués [Lew04].

Enfin cette norme définit également le terme de « capteur virtuel » car sa principale caractéristique est d'être indépendant du réseau dans lequel il se trouve. Pour réaliser cette condition, le capteur n'est en réalité constitué que par l'élément de détection/mesure, le conditionneur de signaux, le convertiseur analogique-numérique et le processeur du signal numérique que l'on peut schématiser comme à la Figure 3 qui est une modification du schéma présent dans [Lew04].

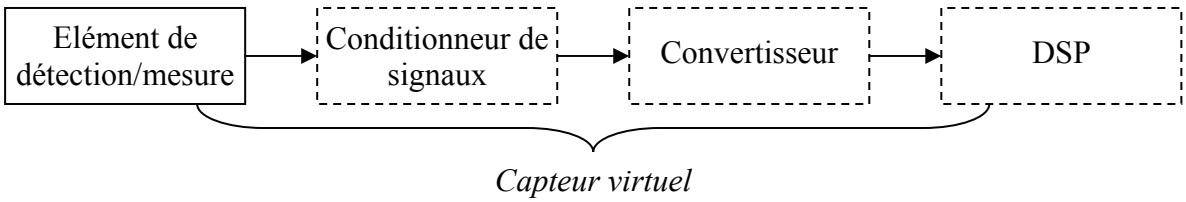


Figure 3 : Modèle de capteur virtuel, inspiré de [Lew04]

II.1.3) La mise en réseau

Les types de capteurs que nous venons de décrire servent de pierre angulaire à l'édification de réseaux dits de capteurs sans-fil. D'autres éléments viennent compléter ce réseau. Il s'agit des *stations de base* et des *collecteurs*. Les collecteurs sont un autre type de noeud du réseau qui permet de rassembler des données provenant de plusieurs *motes* et qui possède également des fonctions de passerelle, reliant ainsi les capteurs avec un réseau externe – Internet par exemple. Ces derniers possèdent beaucoup plus de capacités que les motes tant au niveau de la mémoire que de la vitesse de traitement ou des réserves en énergie. La station de base, quant à elle, est l'élément recueillant les données et/ou les analyses du réseau et servant également à son administration. Un réseau de capteurs sans-fil peut alors ressembler au schéma de la Figure 4 :

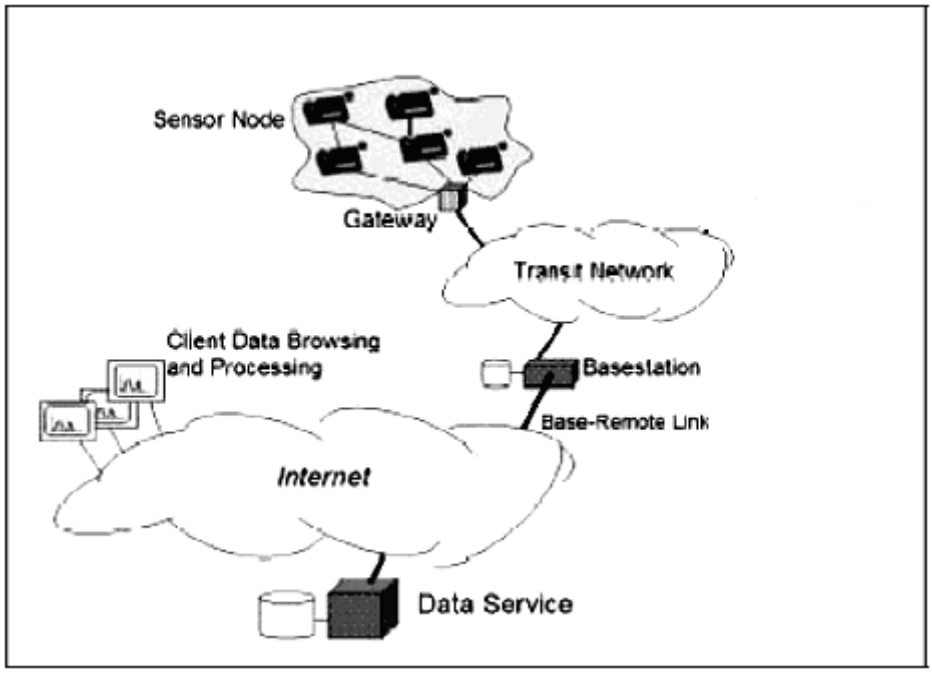


Figure 4 : Schématisation d'un réseau de capteurs sans-fil

Ainsi les données détectées vont parcourir le réseau de capteurs en capteurs jusqu'à la station de base par exemple. En retour, celle-ci peut demander l'exécution d'une tâche particulière à un capteur donné.

Ces réseaux sont utilisés dans des domaines divers nécessitant l'acquisition d'informations concernant l'environnement. Le paragraphe suivant décrit les différentes utilisations possibles d'un *WSN*.

II.1.4) Les différentes utilisations des réseaux WSNs

Comme beaucoup de technologie, le développement des *WSNs* a été suscité par des besoins militaires. En effet, les armées souhaitent être en mesure d'espionner discrètement leurs ennemis. L'absence de câbles entre les noeuds, leur faible taille, le nombre élevé de *motes* déployables – pour couvrir une zone étendue - répondent à ces critères. Ainsi plusieurs applications ont été réalisées dont le SOund SUrveillance System ou le Distributed Sensor Network [KDM05].

Puis, la diminution des coûts fabrication des capteurs ainsi que la réduction de leur taille a entraîné une utilisation dans des applications civiles. Ils peuvent par exemple être utilisés à des fins de surveillance environnementale. En effet, des capteurs peuvent être placés dans des régions glaciaires ou tropicales afin de suivre de manière précise les effets du réchauffement de la planète, les changements climatiques ou l'augmentation de la pollution [KDM05]. Ils peuvent également être employés pour une surveillance de l'habitat car leur déploiement, par exemple en montagne, pourrait permettre de recenser les animaux fréquentant un territoire donné. Des applications industrielles peuvent également les adopter. Une idée d'utilisation pourrait être de placer ces instruments à des points spécifiques, par exemple aux points d'usure des machines ce qui permettrait d'émettre des alertes lorsque leur état général se dégrade [CES04]. Enfin, certains envisagent d'implanter des senseurs dans le corps humain. Ce qui permettrait de contrôler l'état de santé de patients et ainsi d'adapter leur traitement, de prévenir de la dégradation de leur état de santé et par conséquent d'être capable d'anticiper une hospitalisation en urgence [LM&all04]. Mais ceci n'est pour l'instant qu'une utilisation future qui dépend encore grandement des progrès à venir de cette technologie.

II.1.5) Les caractéristiques des réseaux de capteurs sans-fil

Les réseaux de capteur sans fil forment un domaine de recherche éclectique nécessitant la mise en commun de connaissances provenant de nombreuses disciplines différentes telles que l'électronique, la physique, les réseaux, la programmation, etc.

Premièrement, l'électronique fournit des innovations au niveau du matériel. En effet et conformément à la seconde loi de Moore qui dit que « *the number of transistors on integrated circuits doubles every 18 months* », on constate que les composants électroniques disponibles sont régulièrement améliorés. Il existe donc une grande hétérogénéité du matériel disponible. Le Tableau 1 résume les améliorations dont les motes ont bénéficié en quelques années au niveau de la cadence de leur processeur (la vitesse à laquelle les opérations peuvent être effectuées), de la mémoire Flash disponible pour les applications ou de la mémoire utilisable pour les calculs (RAM).

Mote	WeC	rene	dot	mica	mica2	iMote	btNode
Date	1999	2000	2001	2002	2003	2003	2003
Processeur (MHz)	4				7	12	7
Flash (code, ko)	8	8	16	128	128	512	128
RAM (ko)	0,5	0,5	1	4	4	64	4

Tableau 1: Evolution des capacités matérielles des motes [LMGPSWBC04]

En ce qui concerne les réseaux, les différentes techniques existantes ne peuvent pas être appliquées directement. Celles-ci doivent être adaptées afin de prendre en compte les fortes contraintes, notamment celles concernant l'énergie disponible. Par exemple, pour les communications, l'utilisation d'acquittement doit être utilisée avec parcimonie [YB05]. En effet, même si l'utilisation d'une telle technique permet d'augmenter la fiabilité des échanges entre les nœuds, cela engendre des communications supplémentaires et par conséquent une surconsommation d'énergie.

De même, le déploiement d'un tel réseau génère de nouveaux problèmes. Il existe, en effet, plusieurs moyens de déployer un WSN sur une zone à surveiller. Cette dissémination des capteurs peut être réalisée aléatoirement, par exemple, en « parachutant » des motes depuis un avion ou un bateau. Dans ce cas, il devient alors impossible de déterminer une configuration a priori du réseau. En effet, certains pourraient atterrir à l'envers ce qui les empêcheraient non seulement de faire leur travail de détection mais également de communiquer avec les autres éléments du réseau. Pour remédier à ce problème, une auto-organisation de ce dernier s'avère nécessaire c'est-à-dire que les *motes* doivent savoir localiser les nœuds voisins et établir les routes pour que l'information puisse circuler à travers le réseau. Un deuxième type de déploiement consiste à placer « à la main » des capteurs dans des endroits stratégiques mais cette solution devient impensable dès que l'on considère un très grand nombre de capteurs. Enfin, un mélange de ces deux techniques peut être envisagé.

De nouvelles techniques de routages doivent également être mises en place. En effet, dans les réseaux dits « classiques » comme les LANs, Internet, etc, les paquets sont routés via l'adresse IP du destinataire. Or dans un réseau WSN, il peut arriver qu'on s'intéresse moins à l'auteur d'une donnée qu'à la donnée elle-même. Par exemple si on reprend l'exemple du thermomètre, cela revient à se demander : « Y a-t-il des endroits où il fait plus de 20°C ?» et non « Où fait-il plus de 20°C ? ». Cette technique particulière est appelée « basé-attribut ».

De plus, on peut envisager qu'un utilisateur traverse le « champ » de motes afin de collecter des informations ou de reconfigurer le réseau. Il convient alors de fournir à cet utilisateur des moyens d'administrer aussi bien un simple mote que le réseau dans son ensemble. C'est dans ce cadre que s'inscrivent les recherches actuelles sur la possibilité de définir un middleware pour les WSNs. Elles sont présentées dans la section suivante.

II.1.6) Les challenges/Les besoins

L'utilisation d'un middleware apparaît comme un besoin fondamental pour faciliter l'évolution des réseaux de capteurs. En effet, la description non exhaustive précédente (II.1.4) des différentes utilisations possibles d'un tel réseau démontre la diversité des caractéristiques des applications. En effet, les applications des WSNs sont fortement dépendantes du domaine et des objectifs envisagés. A chaque application correspond une architecture du réseau et une implémentation. Certaines, par exemple, peuvent nécessiter une transmission des données fiables [KDM05] alors que d'autres peuvent se permettrent de perdre des données en cours de routage mais nécessitent une grande précision dans les données collectées. Or les techniques actuelles nécessitent souvent de redévelopper, soit dans le meilleur des cas une partie, soit dans le pire des cas toute l'application. Un middleware peut alors apporter une solution à ce problème en s'adaptant facilement à l'hétérogénéité des applications développables sur de tels réseaux. En effet, dans les systèmes traditionnels, un *middleware* est utilisé pour masquer l'hétérogénéité des systèmes mis en réseau et faciliter le développement d'applications distribuées. De plus, il est défini comme une couche logicielle servant d'intermédiaire entre le système d'exploitation et une application distribuée.

Cependant la conception d'un middleware pour les réseaux de capteurs n'est pas une démarche triviale. En effet, le nombre et la sévérité des contraintes existantes constituent un premier obstacle à son élaboration. De plus, les réseaux de capteurs étant une technologie particulièrement récente, il n'existe pas encore de consensus concernant la définition des interfaces du système d'exploitation. Celles-ci font toujours l'objet de recherche. En outre, de nombreuses applications exécutent des opérations matérielles directement sans passer par un composant système [YB05].

Malgré tout, certaines fonctionnalités récurrentes dans les applications actuelles (agrégation de données, contrôle et gestion du réseau) permettent d'établir des premiers éléments dans la définition d'un tel middleware. Dans ce cadre, des principes de conception ont néanmoins été proposés dans [BHRT04] et [YKP04]. Un middleware doit donc être entre autre :

- Evolutif pour s'adapter aux contraintes ;
- Générique pour s'appliquer à différentes applications possédant des interfaces communes ;
- Adaptatif pour reconfigurer sa structure ;
- Réflectif pour changer de comportement en fonction de l'environnement et des circonstances ;
- Léger.

De plus, des mécanismes configurables en fonction des besoins de l'application devront être proposés. Il devra également exister des mécanismes autorisant l'accès aux données détectées à partir d'application extérieures au réseau ainsi que des communications basées évènement et centrées sur les données.

Nous avons présenté ici le contexte général, les spécificités et les contraintes du domaine dans lequel s'inscrit ce travail, nous allons, dans la section suivante, nous intéresser aux caractéristiques de TinyOS, le système d'exploitation de référence installé sur les capteurs.

II.2) TinyOS et sa distribution

TinyOS [TOS04] est un système d'exploitation open-source spécialement conçu pour les applications embarquées fonctionnant en réseaux et, en particulier, pour les réseaux de capteurs sans-fil. Initialement développé au sein de l'université de Berkeley en Californie, TinyOS est devenu le standard *de facto*. En effet, celui-ci est installé sur tous les motes disponibles actuellement et de nombreux groupes de recherche ainsi que des industriels s'en servent afin de développer et tester des protocoles, différents algorithmes, des scénarios de déploiement [TOS04].

Cette popularité est liée au fait que TinyOS fournit non seulement des outils de développement et de simulation aux développeurs mais également une solution permettant de développer rapidement des applications répondant à la diversité des caractéristiques existantes d'un réseau à l'autre telles que le besoin de fiabilité dans les informations récoltées, la qualité de service du réseau ou encore aux capteurs mis en jeu. En outre, le domaine de l'embarqué impose de sévères contraintes notamment en ce qui concerne l'espace de stockage et l'espace mémoire alloués au système d'exploitation et aux applications tournant dessus. TinyOS répond à ce problème en générant une très petite empreinte mémoire [GLBWBC03], celle-ci correspondant à la fusion du système d'exploitation et de l'application exécutée. Enfin, en ce qui concerne plus particulièrement le domaine des réseaux de capteurs, les capteurs sont souvent à l'origine de la détection d'un phénomène et informe le réseau de son existence. TinyOS propose ainsi un modèle de programmation orienté évènement.

Cette section décrit TinyOS en commençant dans un premier temps par la description de son modèle et des principales notions associées à celui-ci. Ensuite, nous détaillerons le langage NesC utilisé pour l'implémentation de TinyOS. Celui-ci est un langage orienté composant s'inspirant du C classique.

II.2.1) Les notions principales

Afin de fournir les spécificités décrites précédemment, TinyOS est construit autour des différents concepts décrits ci-dessous. Le premier repose sur l'utilisation de composants. Ceux-ci sont constitués d'une *frame*, de *tâches* et d'*interfaces*. La *frame* contient l'état interne du composant. Il s'agit d'un espace mémoire réservé de taille fixe permettant au composant de stocker les variables globales et les données qu'il utilise pour réaliser ses fonctionnalités. Il n'en existe qu'une seule par composant et celle-ci est allouée statiquement à la compilation. Les *tâches* quant à elles contiennent l'implémentation des fonctions métiers, décomposées en deux catégories : les commandes et les évènements. Enfin, les *interfaces* représentent le descriptif de ces fonctions. Un composant TinyOS peut donc être modélisé tel que sur la Figure 5 où les triangles représentent les gestionnaires d'évènements, les triangles inversés le point d'accès des commandes, les flèches pointillées les évènements signalés et les flèches pleines les commandes transmises.

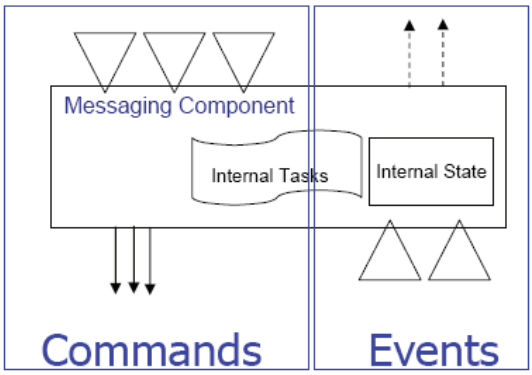


Figure 5 : Schématisation d'un composant TinyOS [HSWHCP00b]

Ces composants s'inscrivent dans le cadre de l'utilisation d'une architecture basée composant qui se révèle spécifique par l'implémentation proposée par nesC. Grâce à celle-ci, il est possible de créer une application par assemblage des seuls éléments strictement nécessaires tant au niveau système qu'au niveau applicatif. A cette fin, TinyOS fournit une réserve de composants systèmes utilisables au besoin. Parmi les plus fréquents, nous pouvons citer ceux concernant les entrée/sorties, les timers, les protocoles communications, etc. Ceux-ci couvrent aussi bien les parties hardware que software d'un capteur. Il ne reste alors au concepteur qu'à créer ses propres composants c'est-à-dire ceux répondant aux fonctionnalités métiers de l'application cible et à les lier avec ceux fournis par TinyOS. Cet assemblage est réalisé indépendamment de l'implémentation.

TinyOS utilise pour cela un *Langage de Description d'Architecture* ou *ADL*[6] afin de définir quels sont les composants impliqués dans la création de l'application ainsi que la manière dont ils sont reliés. Cette liaison entre composants repose sur la notion d'interface. Une interface contient le descriptif des tâches que le composant est capable de rendre ou celles dont il a besoin pour fonctionner correctement. On parle alors respectivement d'interfaces de services fournis et d'interfaces de services requis. Un ADL permet ainsi de schématiser une vue globale de l'application de la même manière qu'un plan d'architecte modélise une maison en décrivant les différentes pièces de celle-ci et la manière dont elles sont disposées et « communiquent » entre elles (portes, couloirs, etc.). En continuant l'analogie, les pièces représentent les composants et les « voies de communication » les liaisons.

Dans TinyOS, ce plan est appelé « graphe de composants ». Il favorise la réutilisabilité des composants et se représente comme dans la Figure 6 [HSWHCP00a]. Les rectangles représentent les composants, les flèches blanches les « interfaces de services fournis », les flèches noires les « interfaces de services requis ». Les traits partant d'un composant vers un triangle noir ou blanc modélisent une liaison.

[6] ADL : Architecture Description Language

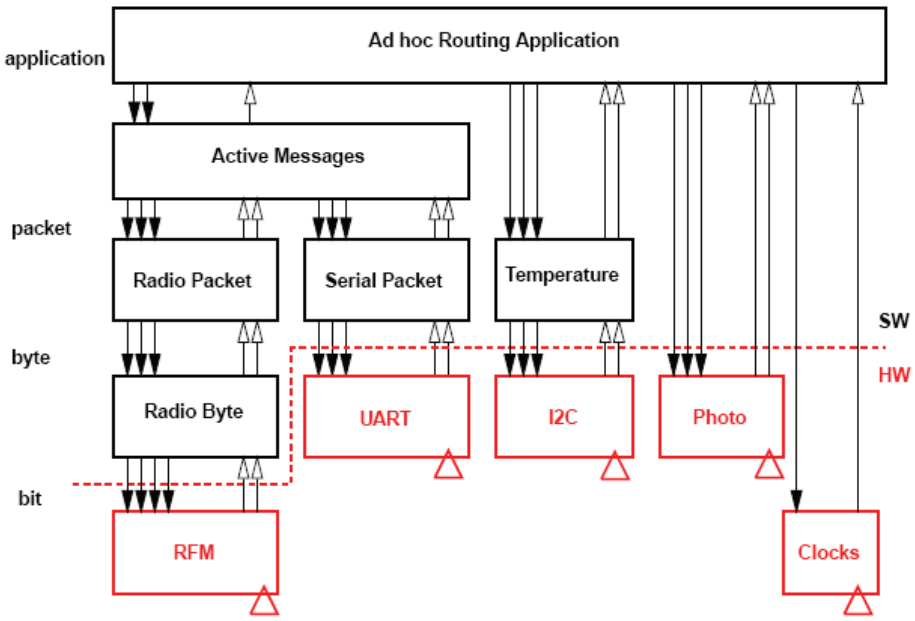


Figure 6 : Graphe de composants d'une application visant au routage des relevés des capteurs [HSWHCP00a]

Le deuxième concept important fixe un modèle mémoire. En effet, une des particularités de TinyOS est d'interdire les allocations dynamiques ainsi que celles se produisant à l'exécution. Pour cela, TinyOS s'appuie sur le graphe de composant précédemment décrit afin de déterminer la taille de chaque composant et ainsi établir statiquement leurs liaisons à la compilation. En outre, afin de conserver l'espace mémoire, les pointeurs de fonctions sont également interdits. En effet, leur utilisation nécessiterait de stocker non seulement la (ou les) fonction(s) mais également le pointeur vers celle(s)-ci. De même, les variables globales permettent d'économiser de la mémoire. En effet, de façon intuitive, il est moins coûteux de ne posséder qu'un seul exemplaire de données plutôt que plusieurs. Ainsi dans TinyOS lorsque plusieurs composants utilisent des données similaires, celles-ci sont déclarées en tant que variables globales et sont stockées dans un espace mémoire réservé à leur effet – ce qui correspond au cadre « Global » de la Figure 7. Les composants requérant leur utilisation y accèdent au moyen d'un pointeur. Enfin, un dernier espace mémoire est réservé pour les variables locales. Il s'agit de l'espace « Stack » présent sur la Figure 7. Celui-ci fonctionne à la manière d'une pile permettant l'empilement et le dépilement de ces variables temporaires. L'empilement faisant diminuer l'espace « Free » et le dépilement l'augmentant. L'utilisation de la RAM d'un mote peut donc être modélisée de la manière suivante :

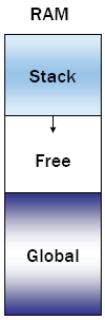


Figure 7 : Modélisation de l'utilisation de la mémoire d'un mote [TOS05]

Le dernier concept fondateur concerne la concurrence. Dans TinyOS, celle-ci peut tout d'abord être générée par des mécanismes calculatoires différés : les *tâches*. Il s'agit de sortes d'appels de fonctions réalisés par les composants. Cela consiste à placer « la fonction » à exécuter selon sa priorité dans l'ordonnanceur de TinyOS, une file FIFO bornée non-préemptive, et à se terminer immédiatement. Lorsque la file est pleine, la tâche ayant la priorité la plus faible est enlevée pour permettre à la nouvelle tâche dont la priorité est plus élevée d'y entrer. Leur tour venu, les tâches alors s'exécutent jusqu'à complétion sauf si elles sont préemptées par un évènement. Il est à noter que les tâches ne peuvent pas se préempter entre elles et qu'elles doivent être courtes. S'il s'agit d'une tâche longue, celle-ci se déroulera sur plusieurs tâches. La deuxième source de concurrence provient des *évènements*, c'est-à-dire d'un phénomène détecté par le capteur, d'un évènement matériel comme une interruption par exemple ou de l'annonce de la terminaison d'une opération *split-phase,* i.e. une tâche de longue durée. Les évènements semblablement aux tâches s'exécutent jusqu'à complétion mais peuvent cependant se préempter entre eux [GLBWBC03], appeler des commandes, poster de tâches ou signaler d'autres évènements. Enfin, les *commandes* sont des opérations similaires aux évènements mais constituent uniquement des points d'entrée pour le composant c'est-à-dire qu'elles ne peuvent seulement appeler que des commandes de plus bas niveau ou poster des tâches. Ce qui peut se résumer comme sur le graphe temporel de la Figure 8 où un évènement est déclenché par une interruption provenant d'un composant matériel.

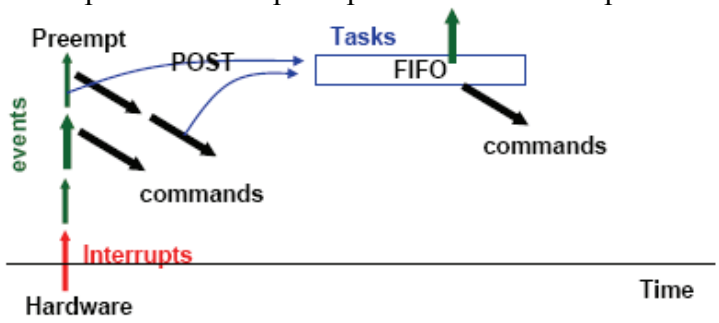


Figure 8 : Modélisation de l'ordonnanceur TinyOS [Liu05]

Pour pouvoir implémenter le modèle TinyOS, un nouveau langage de programmation a été développé en s'appuyant sur le langage C mais en le modifiant afin de prendre en compte les spécificités de ce modèle et les fortes contraintes du domaine. Il s'agit de nesC. Ce langage décrit à la section suivante.

II.2.2) NesC

NesC est un langage de programmation conçu pour la réalisation de systèmes embarqués distribués. Il cible en particulier l'implémentation d'applications pour les réseaux de capteurs. Il offre donc une réactivité importante vis-à-vis de l'environnement, une gestion de la concurrence même intensive et un support de communication. De plus afin de tenir compte des fortes contraintes du domaine visé, de nombreuses optimisations sont proposées pour diminuer l'occupation de l'espace mémoire. En outre, afin de permettre un développement simple, robuste et rapide des applications, nesC s'appuie sur des concepts généraux proposés dans l'ingénierie logicielle basée composant (cf. II.3.1). Il permet notamment la décomposition d'une application en « modules » réutilisables.

Pour obtenir ces différents éléments, nesC fournit d'abord trois abstractions de programmation à savoir les *interfaces,* les *modules* et les *configurations*. Dans les différents exemples de code suivant, les mots-clés du langage sont mis en évidence en gras.

Premièrement, les interfaces spécifient un ensemble de fonctions, appelées commandes, qui doivent être implémentées par le fournisseur de l'interface et un ensemble de fonctions, appelées évènements, qui doivent être implémentées par l'utilisateur de l'interface [GLCB03]. Afin de distinguer les fonctions concernant un évènement de celles concernant une commande, les en-têtes des fonctions sont précédés des mots-clés respectifs *event* ou *command*. Voici un exemple simple d'interface :

```
Interface Timer {
   command result t start(char type, uint32 t interval);
   command result t stop();
   event result_t fired();

}
```

De plus, comme il a été mentionné précédemment, le modèle mémoire fixé par TinyOS n'autorise pas les pointeurs de fonctions. Afin de néanmoins proposer un mécanisme alternatif, nesC utilise des interfaces paramétrées. Celles-ci permettent à l'utilisateur de créer un ensemble d'interfaces identiques et d'en sélectionner une seule à appeler grâce à un identifiant.

```
interface SendMsg[uint8 t id]
```

Ainsi, une instance d'interface paramétrée correspond à de multiples interfaces vers ce composant, une pour chaque tuple distinct de valeurs de paramètres. L'interface SendMsg déclare donc 256 interfaces de type SendMsg [GLCB03].

Les modules sont eux les éléments de base de la programmation. Ils permettent en effet d'implémenter les composants et sont stockés dans un fichier possédant la structure suivante :

```
module NomModuleM {
 provides {
//liste des interfaces fournies, ex :
interface NomInterfaceFournie1 ;
 }
 uses {
     //liste des interfaces requises, ex :
     interface NomInterfaceRequise1 ;
 }
}
implementation {
     //déclarations des variables, ex :
     int state ;

     //implementations des fonctions decrites par les
     //interfaces  fournies ;
}
```

Il est à noter que tous les composants TinyOS doivent posséder l'interface StdControl car celle-ci est utilisée pour initialiser, démarrer et arrêter les composants.

Enfin, les configurations, quant à elles, permettent de décrire les composants composites c'est-à-dire des composants composés d'autres composants c'est-à-dire de granularité supérieure. Une configuration permet donc de décrire l'architecture. Une configuration est donc constituée de modules et/ou d'interfaces ainsi que de la description des liaisons entre ces composants.

```
configuration NomModule {
  }
implementation {
   //liste des modules et configurations utilises, ex :
   components Main,Module1,…,ModuleN,Config1,…,ConfigM ;

   //descriptifs des liaisons
   //Interface fournie <- Interface requise ou
   //Interface requise -> interface fournie, ex :
   Main.StdControl -> Module1.StdControl ;
}
```

Détaillons ici, quelques caractéristiques concernant les configurations. La première d'entre elles concerne une simplification. Lors du descriptif des liaisons, il est en effet possible de ne pas préciser l'interface fournie par un module. Dans ce cas, elle possédera le même nom que celle requise. L'autre caractéristique concerne le composant *Main*. En effet, il est à noter que ce composant est obligatoirement présent dans la configuration décrivant l'ensemble de l'application car son rôle est de démarrer l'exécution de l'application.

En outre, le modèle d'exécution proposé par nesC repose sur les tâches et les gestionnaires d'interruption. Les tâches comme mentionné précédemment, s'exécutent jusqu'à complétion sauf si elles sont préemptées par un évènement et les interruptions sont déclenchées de manière asynchrone par le matériel. Or cela peut conduire à des accès concurrents[7] causés par la présence des variables globales. Pour éviter que des « *races* » ne se produisent, l'accès aux variables globales ne doit se faire qu'à l'intérieur de tâches ou qu'à l'intérieur d'instruction atomique. Ainsi, il existe deux types de codes permettant la gestion de la concurrence : le *code synchrone* et le *code asynchrone*. Le code synchrone est la portion de code d'une fonction (commandes, évènements ou tâches) accessible seulement par des tâches. Le code asynchrone est lui accessible par au moins un gestionnaire d'interruption. Enfin, afin d'assurer qu'aucune situation de compétition ne se produise, nesC impose un « Race-Free invariant » : *Any update to shared state is either Synchronous Code-only or occurs in an atomic statement* [GLCB03]. L'atomicité est implémentée en désactivant ou réactivant les interruptions. Enfin, le système permet également au programmeur de signifier qu'une situation probable de compétition ne se produira pas, cela est réalisé en spécifiant le code correspondant comme *norace*.

[7] en anglais, race conditions

II.2.3) La compilation

La première étape de ce processus consiste à compiler les fichiers nécessaires à l'application et au système d'exploitation. Celle-ci est réalisée via le compilateur nesC fourni par TinyOS. Son rôle est premièrement de transformer les fichiers nesC en fichier C et deuxièmement d'y intégrer les fichiers du noyau de TinyOS. Ce qui permet d'obtenir un fichier source C unique. Une fois cette étape accomplie, il ne reste alors qu'à utiliser un compilateur C traditionnel qui va utiliser le fichier précédemment créé afin de générer une application exécutable. Celle-ci sera donc constituée par la « fusion » système d'exploitation et du code applicatif. Ces différentes phases peuvent être synthétisées comme présenté à la Figure 9 :

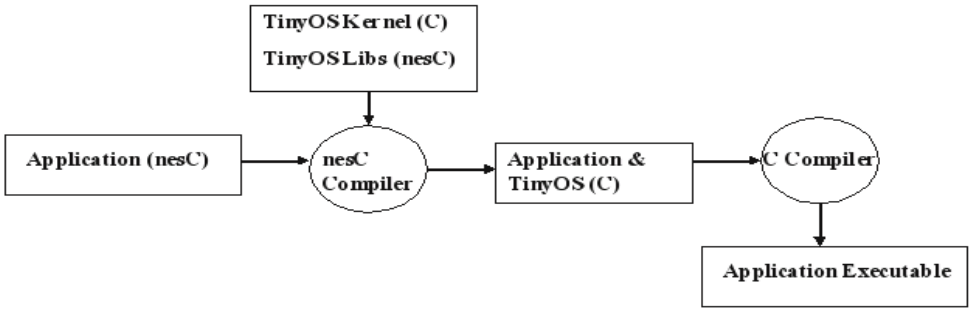


Figure 9 : Les étapes de la compilation d'une application TinyOS[Mau]

L'intégration du système d'exploitation TinyOS, dont le cœur n'occupe que 386 octets, dans l'application est une première optimisation proposée lors de la compilation par nesC afin de préserver l'espace mémoire. En poursuivant le même but, des opérations d'*inlining* sont réalisées. Il s'agit d'un processus consistant à insérer le code d'une fonction à l'endroit où celle-ci est appelée. Dans nesC, l'inlining n'est pas seulement réalisée à l'intérieur d'un module mais aussi entre des modules différents. De même, les *dead codes*, c'est-à-dire les morceaux de codes inutiles telle qu'une fonction qui n'est jamais appelée, sont supprimés.

Après avoir présenté le domaine des réseaux de capteurs sans-fil nous allons nous intéresser à la CBSE dans la section suivante. Il s'agit d'une approche visant à pallier les limites de la technologie orientée objet pour le développement d'applications pouvant s'adapter rapidement et à un moindre coût aux nécessaires évolutions du marché.

II.3) Etat de l'art sur la CBSE

Nous allons dans cette section dresser un rapide aperçu de l'ingénierie logicielle basée composant plus connue sous le nom de Component-Based Software Engineering (ou CBSE).

II.3.1) Les concepts fondamentaux de la CBSE

La CBSE repose sur une série de concepts maintenant clairement établis. Nous les présentons ici.

II.3.1.1) Les composants

Bien que la notion de composants soit au cœur de la CBSE, il n'en existe pas pour autant de définition universelle. Cependant, celle la plus souvent retenue est celle proposée par Szyperski [SGM02] : *« A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties ».* Autrement dit, un composant possède trois caractéristiques fondamentales. La première d'entre elle est qu'un composant est semblable à une boîte noire c'est-à-dire qu'il masque totalement les détails de son implémentation et ne communique que par le biais de ses interfaces. La deuxième spécificité est la prédisposition des composants leur permettant d'être assemblés dans le but de développer d'autres composants ou de construire des applications toutes entières. La dernière propriété a trait au déploiement. En effet, un composant peut non seulement être déployé par une personne qui ne l'a pas conçu mais également indépendamment de tout autre élément. On peut alors parlé d'entité autonome.

Afin de simplifier, nous allons réaliser une analogie avec un puzzle, un composant étant ainsi équivalent à une pièce de ce puzzle. Chacune possède un fragment de l'image finale ainsi qu'un contour qui est lui propre mais pouvant s'adapter au contour d'une autre pièce. Si deux pièces possèdent un contour opposé, un assemblage est possible. Dans le cadre de l'ingénierie logicielle basée composant, on parle de composition. Les bords « représentent » les interfaces contractuelles et le fragment d'image correspond à l'implémentation de ce composant. Enfin, une pièce peut demeurer seule sur le plateau c'est l'autonomie et être finalement installé par un autre amateur de puzzle.

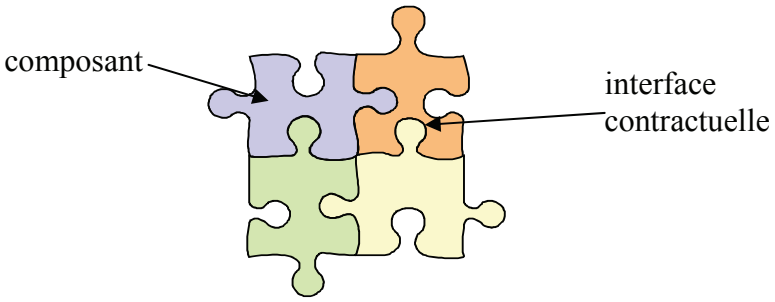


Figure 10 : Illustration de la notion de composant

Enfin, un composant ne possède pas une taille fixe. Il peut représenter un objet classique comme un composant contenant d'autres composants contenant eux-mêmes d'autres composants et ainsi de suite. On parle alors de granularité arbitraire du composant.

II.3.1.2) Les interfaces

Afin que deux composants puissent collaborer, i.e. s'échanger des données ou se demander mutuellement l'exécution d'une tâche, il est nécessaire de les relier. Pour cela, on utilise les points d'accès aux composants : les interfaces. Une interface se présente sous la forme d'une collection de déclarations d'en-tête de fonctions. Elles décrivent donc de manière exhaustive l'ensemble des services que le composant est capable de rendre ou l'ensemble de ceux qu'il requiert pour fonctionner correctement. Cette distinction est représentée en UML par la notation suivante :

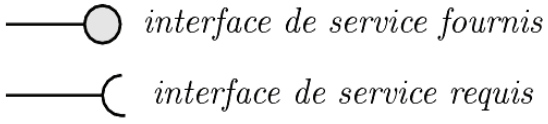


Figure 11 : Représentation UML des 2 types d'interfaces existantes

La connexion d'une interface de service requis d'un composant avec une interface de services fournis par un autre composant forme alors un canal de communication entre ces deux composants. Mais une telle *liaison* n'est établie qu'à condition que l'interface de services fournis offre effectivement toutes les opérations pouvant être appelées par l'interface de services requis. On peut également parler de connexion par analogie avec le modèle/serveur. Cela est modélisé en UML de la manière suivante :

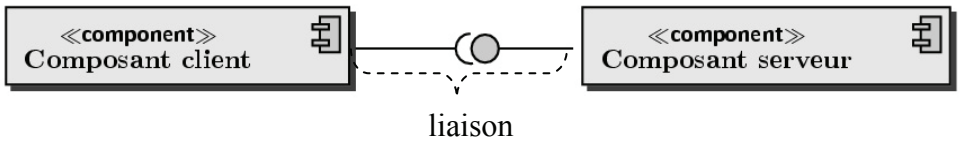


Figure 12 : Schéma UML 2.0 d'une liaison

Ces assemblages peuvent être réalisés au moyen d'un ADL (cf. II.2.1).

La section suivante décrit un modèle reprenant les concepts établis par cette approche.

II.3.2) Le modèle initial de Think

Le modèle Think a été développé par Jean-Philippe Fassino au cours de sa thèse [Fas01]. Celle-ci, soutenue en 2001, propose un modèle permettant de représenter la diversité des architectures des systèmes flexibles. Il est en effet possible, grâce à Think, de modéliser aussi bien un système d'exploitation, réparti ou non, qu'un intergiciel. Afin de couvrir cet espace de conception, Think ne fait pas de choix d'implémentation et laisse aux concepteurs de systèmes le choix des compromis satisfaisants les contraintes des applications cibles. Il vise ainsi à la modélisation uniforme des systèmes d'exploitation et des infrastructures réparties [Fas01].

Ce modèle a été conçu pour répondre aux besoins de maintenance et d'évolutivité de ces systèmes. Considérons, par exemple, l'utilisation de noyaux monolithiques pour les systèmes d'exploitation. Cette politique est souvent peu adaptée aux applications s'exécutant dessus car dans ce cas tous les appels systèmes potentiellement requis par un ensemble d'applications cibles sont proposés. Pourtant, si on se place dans un domaine spécifique ou si on sait qu'un seul type d'application sera exécuté, alors seulement une partie de ces appels système sera effectivement utilisée. Ainsi, Think permet au concepteur de systèmes de ne sélectionner que les aspects essentiels pour les applications cibles. Cette approche est particulièrement intéressante pour le monde des systèmes embarqués, et par extension pour celui des réseaux de capteurs, où l'élimination du superflu est un objectif capital afin de minimiser l'utilisation des ressources.

Cette section fournit une description non exhaustive du modèle Think en commençant premièrement par une présentation des concepts distinctifs de ceux présentés dans la section II.3.1) sur l'ingénierie basée composant (composants, interfaces, liaisons). Puis nous présentons la notion de domaine ainsi que la bibliothèque de composants, i.e. Kortex. Nous terminons cette section en expliquant les différentes étapes conduisant à la génération d'une image exécutable.

II.3.2.1) Les notions principales

L'architecture proposée dans [Fas01] repose notamment sur l'utilisation de composants, de liaisons et de domaines. Les paragraphes suivants explicitent ces différentes notions.

II.3.2.1.1) Les composants

Le modèle Think repose sur la notion de composant telle que définie précédemment c'est-à-dire qu'il réalise les tâches pour lesquelles il a été construit et qu'il les propose à d'autres via le biais d'interfaces.

Think vise à limiter les contraintes imposées au concepteur de système ainsi la granularité des composants n'est pas imposée et de même, le modèle Think ne fait pas de supposition concernant le langage utilisé pour programmer ces éléments. Un langage de bas niveau tel que l'assembleur ou le C peut-être employé aussi bien qu'un langage de haut niveau comme le C++ ou Java. Il est à noter qu'à l'heure actuelle, seuls l'assembleur et le C sont utilisés et que le C++ est en cours de développement.

Enfin, dans Think, la notion de composant est utilisé jusqu'au plus bas niveau. Autrement dit, même les éléments matériels sont définis sous forme de composants. Cependant, ceux-ci ne fournissent que les fonctionnalités proposées par le hardware afin de rester conforme avec la philosophie de Think qui est de n'imposer aucune abstraction système. Ces composants étant fortement liés au matériel sous-jacent, les systèmes les utilisant ne seront donc portables que sur des machines strictement similaires.

II.3.2.1.2) Les interfaces

Le mécanisme utilisé pour définir des interfaces conformes aux concepts proposés par l'ingénierie logicielle basée composant repose sur l'utilisation de descripteurs. Ceux-ci permettent de séparer la description de l'implémentation. Ainsi le descripteur d'interface, nommé `meth`, contient un pointeur vers une table contenant des structures correspondant aux méthodes. Le premier élément de cette structure correspond à l'en-tête de l'opération et le deuxième à un pointeur vers son implémentation. La Figure 13 schématise donc cette description.

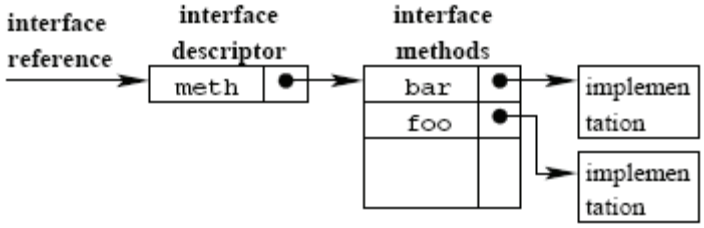


Figure 13 : Représentation d'une interface à l'exécution [FSLM02]

Cependant, cette représentation correspond à une première optimisation et n'est permise que lorsqu'il n'existe qu'un seul composant possédant cette interface. En effet, selon la nature du composant cible, des améliorations sont envisagées afin de minimiser par exemple la consommation de mémoire et/ou les coûts engendrés par l'allocation. Premièrement lorsqu'un composant possède exactement une seule interface, celle-ci pouvant servir de points d'accès à plusieurs composants, alors les données sont stockées directement dans le descripteur. On obtient alors la Figure 14 suivante :

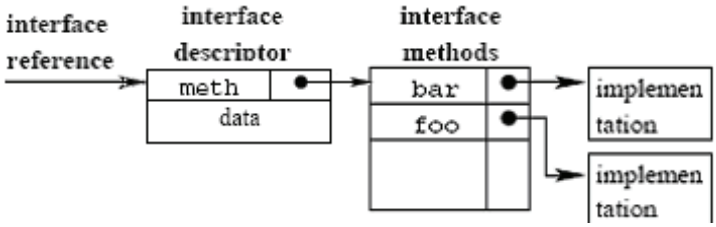


Figure 14 : Représentation de l'interface unique d'un composant [FSLM02]

Enfin dans tous les autres cas, le descripteur devient une table de description dont la deuxième ligne, nommée `data`, contient un pointeur vers les données comme cela est présenté sur la Figure 15 :

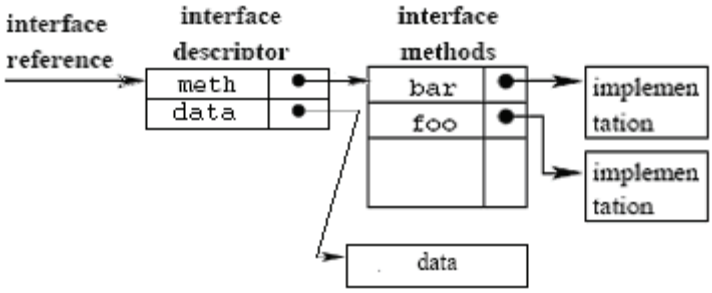


Figure 15 : Représentation d'une interface dans le cas général [FSLM02]

II.3.2.1.3) Les liaisons

Le modèle Think s'appuie sur la notion de *liaison flexible* qui permet de modéliser de façon uniforme l'ensemble des interactions entre les différents composants du système [Fas01]. Derrière ce concept, différents types de liaison peuvent ainsi être envisagés et créés dynamiquement comme par exemple la sérialisation qui permet l'échange d'objets binaires ou l'interposition qui correspond à un « détournement » de liaison permettant entre autre de comptabiliser les appels émis via cette liaison. La création des liaisons est réalisée grâce à un composant usine particulier appelé *usines à liaisons*.

Cependant les liaisons ne relient pas forcément des composants présents physiquement sur une même machine. Ils peuvent en effet être distants. Les appels de services sont alors transmis par des RPC[8] à travers un réseau tel que Ethernet par exemple. Il s'agit ici d'un point sur lequel il convient de s'attarder. En effet, une des caractéristiques des réseaux de capteurs sans-fil est de ne justement pas posséder de réseaux câblés. Ainsi tous les appels distants passent par l'émetteur-récepteur. Or les opérations utilisant cet élément, en particulier pour des émissions de données, consomment énormément d'énergie. Malheureusement, il s'agit là d'une des ressources critiques du système. Il convient donc de n'utiliser ces appels distants qu'à bon escient ou de trouver un nouveau moyen d'implémenter ces appels distants par exemple en modifiant la forme du paquet réseau.

II.3.2.1.4) Les domaines

Le concept de domaine, quant à lui, permet de modéliser l'organisation hiérarchique d'un système [Fas01] c'est-à-dire qu'un domaine peut-être constitué d'autres domaines s'exécutant en parallèle. Un domaine est ainsi constitué de composants formant le *contenu* et d'un composant particulier, le *conteneur*.

Le domaine propose en outre des aspects non-fonctionnels qui servent à l'administration et à la manipulation des composants du contenu. Ce dernier est par exemple en charge de l'exécution, de la configuration, de la maintenance des composants ainsi que de la surveillance de la sémantique du domaine à la manière d'un contrôleur dans le modèle de composant Fractal (cf. II.3.3.1).

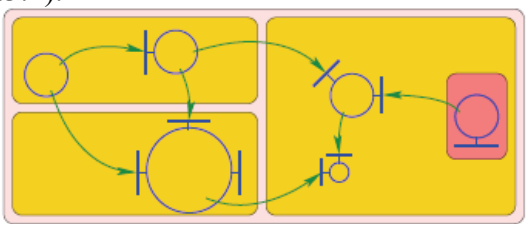


Figure 16 : Système à base de composants, de liaisons et de domaines [Fas01]

Il existe trois classes de domaine : le *domaine matériel*, le *domaine superviseur* et les *domaines applicatifs* [Fas01]. Le domaine matériel recouvre l'ensemble des ressources matérielles d'une machine (processeur, mémoire, périphériques, etc.). Le domaine superviseur permet l'accès à ces ressources et il peut être vu comme le noyau d'un système d'exploitation d'un système classique. Enfin les domaines applicatifs correspondent aux composants n'accédant pas directement aux ressources.

II.3.2.2) Kortex

Pareillement à TinyOS, Think fournit un ensemble de composants systèmes fréquemment utilisés au travers d'une bibliothèque nommée Kortex. Ces composants fournissent leurs services indépendamment de l'implémentation d'autres composants. Cela permet de substituer un composant par un autre sans avoir à se soucier des dépendances, la condition *sine qua non* étant que ce dernier possède exactement les mêmes interfaces. Parmi

[8] abbreviation en anglais de Remote Procedure Call signifiant appel de procédure distante

les services proposés, nous pouvons citer entre autre l'ordonnancement, la gestion de la mémoire (plate ou paginée), la gestion réseau ou encore le chargement dynamique de composants. Enfin il est à noter que Kortex ne dispose actuellement que de composants pour PowerPC, Arm ou Unix et qu'une liste des composants fournis pour PowerPC est consultable dans l'Annexe A.

II.3.2.3) La compilation

Afin de construire et assembler les différents éléments nécessaires au système d'exploitation et à l'application tournant dessus, Think utilise la chaîne de construction présentée à la Figure 17. Celle-ci autorise une séparation entre l'aspect architecturel et fonctionnel. Ainsi les langages de description d'architecture et d'interface (respectivement les cadres .adl et .idl de la Figure 17) sont utilisés pour déterminer les composants en présence. Ces descriptions sont ensuite converties en fichiers C et utilisées en complément des implémentations respectives de chacun des composants pour obtenir les fichiers binaires (.o) grâce au compilateur C. Puis l'éditeur de liens, i.e. l'élément linker de la Figure 17, génère à partir de ces fichiers une image correspondant au noyau du système d'exploitation et à l'application tournant dessus (l'élément *kernel* de la Figure 17) et ce dernier sera chargé sur la machine grâce au *boot loader*. Enfin les composants, ou une nouvelle application, sera chargée via le *dynamic loader*.

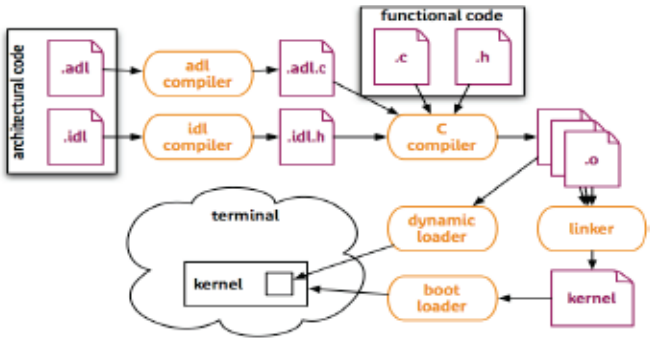


Figure 17 : La chaîne de construction de Think [Lob]

D'autre part, il existe le modèle Fractal qui est modèle de composants générique, modulable et extensible pouvant être utilisé avec différents langage de programmation pour concevoir, implémenter, déployer et reconfigurer différents systèmes [BCS04]. Les concepts clés de ce modèle sont brièvement présentés dans la section suivante. Cependant des informations complémentaires peuvent être consultées dans [BCS04] et [SH05].

II.3.3) Fractal

Fractal est un modèle de composant issu d'une collaboration entre l'INRIA et France Télécom R&D et développé depuis 2002 au sein du consortium ObjectWeb[9]. Il s'agit d'une norme définissant non seulement les divers éléments abstraits constituant le modèle mais également la manière dont ils peuvent être assemblés et interagiront entre eux. Il vise notamment à la création de systèmes et de middleware en s'appuyant sur les concepts de l'ingénierie logicielle basée composant notamment les composants, les interfaces, les liaisons, etc.

[9] http://www.objectweb.org

Ce modèle est aussi fortement orienté vers la modularité et l'adaptabilité aux besoins spécifiques qui peuvent apparaître dans le développement d'applications orientées composant. En effet, il n'impose aucun langage de programmation particulier pour créer, déployer et maintenir des applications. Ce modèle peut également être facilement étendu et s'adapter pour intégrer d'autres modèles de composant tels qu'EJB, CCM, OSGI.

De plus, le modèle de composant Fractal s'appuie sur le principe de la séparation des aspects[10]. La séparation des aspects consiste à dissocier les différentes fonctionnalités (ou aspects) d'une application en différents petits programmes indépendants les uns des autres. En particulier, le modèle de composant Fractal utilise trois aspects :

- la séparation de la conception et de l'implémentation ;
- la programmation orientée composant ;
- l'inversion du contrôle c'est-à-dire que les composants Fractal utilisent une entité externe et séparée afin d'assurer la configuration et le déploiement des composants. [BCS04]. Dans le modèle de composant Fractal, cette fonctionnalité est gérée à l'aide du contrôleur. Ceux-ci sont expliqués dans la section II.3.3.1) suivante.

II.3.3.1) Les composants Fractal

Un composant Fractal est décomposable en deux parties distinctes : une partie non-fonctionnelle et une partie fonctionnelle. Cela correspond respectivement au contrôleur et au contenu [BCS04]. Ainsi, un composant Fractal peut être modélisé tel que sur la Figure 18.

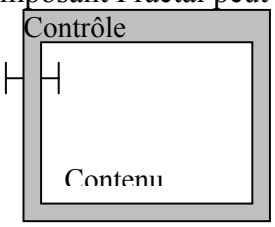


Figure 18 : Vue interne d'un composant

Le contrôleur sert au contrôle et à l'administration des composants. Ainsi différents types de contrôleur sont fournis :

- Le Component : Utiliser pour permettre l'introspection d'un composant.
- Le ContentController : Servant à l'ajout (respectivement le retrait) de sous-composants dans un composant. Ce dernier est alors nommé composant composite.
- Le BindingController : Permettant d'établir ou de supprimer des liaisons entre des interfaces clientes et des interfaces serveurs.
- L'AttributesController : Offrant un moyen d'obtenir et de modifier les « attributs » du composant. Un *attribut* est une propriété configurable du composant [BCS04] comme le prix d'une boisson pour un distributeur de boisson ou la couleur d'un bouton pour une interface graphique.
- Le LifeCycleController : Utiliser pour démarrer et arrêter proprement un composant, ceci afin d'éviter par exemple la perte de données ou que l'état de l'application ne soit plus valable.

[10] en anglais : *« separation of concerns »*

Le contenu contient l'implémentation et les structures nécessaires à la réalisation des requêtes sur les interfaces du composant. En résumé, il contient les divers éléments qui permettent aux composants d'exécuter les tâches pour lesquelles il a été conçu.

Parmi les éléments constituant le contenu, nous pouvons retrouver un ou plusieurs composants, appelés sous-composants. Eux-mêmes pouvant être également composés de sous-composants. Le niveau d'imbrication des composants n'est pas précisé par le modèle : il est arbitraire.

II.3.3.2) Les interfaces Fractal

Comme nous l'avons dit précédemment une interface est un point d'accès au composant. Afin de pouvoir identifier ce point d'accès et ce qu'il permet de faire, une interface possède un nom unique, un ensemble d'opérations pouvant être réalisées via cette interface et un rôle (client, serveur ou contrôle).

Cependant certains concepts présentés dans l'ingénierie logicielle basée composant (cf. II.3.1) correspondent à une appellation différente dans le modèle Fractal. Ainsi le concept d'interface de service requis (respectivement fournis) est nommé interface cliente (resp. serveur). Quant aux interfaces de contrôle, celles-ci sont utilisées pour décrire les fonctionnalités proposées par un contrôleur.

Enfin, la modélisation de ces éléments possède une convention particulière. Les interfaces clientes sont en effet placées à droite, les interfaces serveurs à gauche et les interfaces de contrôle en haut sur la représentation d'un composant comme le montre la Figure 19.

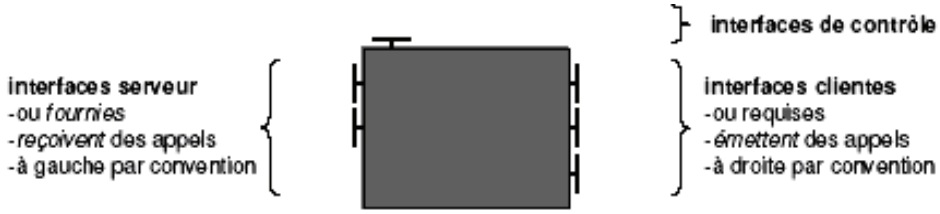


Figure 19 : Vue externe d'un composant [Bru03]

Enfin il est à noter qu'afin de distinguer les différentes interfaces et savoir ce qu'elles sont capables de faire, toutes les interfaces d'un composant doivent posséder un nom unique.

La section suivante résume les éléments résultant de l'union des deux modèles précédemment décrits à savoir Think et Fractal. Afin de lever toutes ambiguïtés, nous avons choisi de baptiser ce modèle Think-Fractal. Dans les sections III, IV et I, les références au modèle Think renvoient en réalité au modèle Think-Fractal.

II.3.4) Le modèle Think-Fractal

Le modèle Think, présenté en 2001, précède de quelques mois le lancement par l'INRIA et France Télécom R&D du projet Fractal qui débute, lui, en janvier 2002. Au départ, les deux modèles sont développés en parallèle mais rapidement Think est intégré au sein de Fractal afin de fournir à ce dernier une branche orientée vers le domaine de l'embarqué et vers le monde des systèmes d'exploitation. Cela est rendu possible grâce à la nature du modèle Fractal qui se veut ouvert et extensible et qui est composé d'une hiérarchie de modèle.

Le premier point qu'il convient de mentionner est que tous les autres éléments du modèle initial Think sont conservés à l'exception de la notion de domaine et de la modélisation. En effet, le domaine disparaît au profit du concept de contrôleur et la modélisation adoptée pour les différentes notions est celle utilisée dans Fractal. En outre, tous les contrôleurs présents dans Fractal, i.e. le LifeCycleController, le BindingController, le ContentController, le Component et l'AttributeController, ont été insérés dans Think-Fractal. Les chargeurs dynamiques et de boot sont toutefois conservés mais sous forme de contrôleurs.

En résumé, nous avons, dans cette section, dévoilé les caractéristiques des réseaux de capteurs sans-fil en insistant notamment sur les contraintes des capteurs ainsi que sur les concepts inhérents à TinyOS le système d'exploitation de référence. Nous avons également détaillé le modèle basé composant de Think et par conséquent présenté le domaine de l'ingénierie logicielle basée composant ainsi que le modèle Fractal qui a intégré Think en tant qu'implémentation pour les systèmes embarqués. Cela nous as permis de dégager les concepts fondamentaux sur lesquels reposera l'étude formulée dans la section suivante. Celle-ci nous permettra de mettre en évidence les éléments nécessaires à la conception d'une architecture logicielle pour les réseaux de capteurs. Nous regarderons en particulier si Think-Fractal peut servir d'alternative afin de pallier les lacunes du modèle proposé par TinyOS notamment ce qui concerne les aspects dynamiques.

III. Vers une architecture logicielle pour les réseaux de capteurs

Le cadre d'étude ayant été établi dans la section précédente, nous allons présenter ici les principes de conception que nous envisageons pour un nouveau modèle de système d'exploitation. Celui-ci permettra de corriger une des limites de TinyOS à savoir l'absence de reconfiguration dynamique. En premier lieu, nous expliquons donc les limites du modèle TinyOS.

III.1) Les limites de TinyOS

Dans le but de réduire la taille des systèmes générés, TinyOS effectue des optimisations et impose des restrictions. Premièrement, toutes les allocations mémoire sont établies statiquement dès la phase de compilation. Deuxièmement, le modèle d'implémentation fournit via NesC ne dispose pas de pointeurs de fonctions. En conséquence, aucune allocation dynamique n'est possible.

Cela représente donc une des limites de TinyOS. En effet, une des caractéristiques pouvant être requise par un réseau de capteur est l'adaptativité aussi bien au niveau du réseau en lui-même qu'au niveau d'un de ses nœuds. La reconfiguration dynamique entre dans ce cadre. Une étude [KNEKLM04] a d'ailleurs mis en évidence les difficultés inhérentes à la mise-en-place de cette propriété sur des motes équipées avec TinyOS. Même si une solution a été proposée, celle-ci s'est révélée lourde à cause des mécanismes employés afin de pallier la nature statique du système et plus particulièrement l'utilisation des composants intermédiaires pour gérer les réaffectations des liaisons. Cette étude propose notamment d'utiliser un système d'exploitation moins contraignant et permettant l'allocation dynamique.

D'autre part, lors de la génération de l'image système, la notion de composant disparaît. Ainsi, si on considère toujours la propriété de reconfiguration et si on admet sa faisabilité alors cela implique que si l'on ne souhaite modifier l'implémentation que d'un seul composant, le protocole de communication par exemple, il est alors impossible de ne remplacer que ce composant. C'est l'image entière du système présente sur le capteur qui doit être réinstallée.

Cela met en évidence un autre problème du modèle. En effet, dans les réseaux de capteurs, les motes servent de nœuds relais pour la propagation des données. Or il est évident qu'une image entière du système possède une taille supérieure à celle d'une partie. Ainsi si on admet encore l'existence d'un mécanisme de reconfiguration pour un mote fonctionnant sous TinyOS, cela implique de découper cette image du système en de nombreux paquets pour la transmettre au capteur ciblé. Cela engendre premièrement une augmentation du trafic sur le réseau et deuxièmement une utilisation accrue du module de communication de chacun des nœuds relais qui vont transmettre l'image d'un mote à l'autre jusqu'au capteur de destination. Par conséquent, chacun des nœuds impliqués dans ce processus de transfert consomme davantage d'énergie. Or il s'agit là d'une ressource limitée.

Par ailleurs, TinyOS ne fournit aucun mécanisme de protection mémoire. Cela implique que le système est particulièrement vulnérable aux crashs et corruptions de la mémoire.

En conclusion, le modèle TinyOS tel que proposé actuellement ne permet pas de répondre aux besoins futurs des applications utilisant les réseaux de capteurs. En effet, les besoins de services avancés tels que la reconfiguration dynamique ou le déploiement de nouveaux services sont désormais reconnus comme un besoin important dans les applications du futur. Les nombreux travaux portant sur l'autonomic computing [KC03] en sont la preuve.

Nous allons donc étudier la possibilité d'utiliser une approche à base de composants pour le développement des applications utilisant des réseaux de capteurs. La plate-forme logicielle Think-Fractal proposée par l'INRIA et France Télécom R&D servira de base à cette étude car premièrement elle a été développée pour répondre aux contraintes des systèmes embarqués et deuxièment, il a été établi que les modèles EJB, CCM et OSGI étaient trop lourds pour les WSNs [SGVVRF04]. Nous nous situons ainsi dans le cadre de la proposition faîte dans [KNEKLM04].

III.2) Vers un système d'exploitation basé composants

Il a été établi dans la section précédente que le modèle proposé par TinyOS est trop rigide pour fournir des services avancés tels que par exemple la reconfiguration dynamique. Or malgré une consommation de ressources notamment mémoire un peu excessive, l'utilisation du modèle proposé par Think[11] semble prometteuse plus particulièrement en vue d'ajouter les aspects dynamiques manquants au modèle TinyOS. L'objectif de cette section est donc de présenter la conception d'un modèle de système d'exploitation pour capteurs associant les aspects éprouvés de TinyOS mais également ceux de l'ingénierie logicielle basée composant au travers de Think.

La première caractéristique de TinyOS qu'il nous semble utile d'intégrer dans Think est le modèle d'exécution basé évènement. En effet, un tel modèle est fréquemment utilisé dans les systèmes embarqués car il permet premièrement de générer une petite empreinte mémoire et deuxièmement de pouvoir contrôler plus facilement l'ordonnancement des activités. Une telle intégration est possible car une des philosophies de Think est justement de permettre la réalisation de n'importe quel type de système d'exploitation et par conséquent celle d'un système d'exploitation basé évènement. Mais cela pose de nombreuses questions à commencer par la façon de représenter un système d'exploitation sous forme de composants.

Pour y répondre, nous allons tout d'abord présenter les différents éléments matériels constituant le capteur ainsi que la manière dont ils interagissent entre eux. Comme il a été décrit dans l'état de l'art, un capteur possède un microprocesseur, des périphériques (réseau, capteur, Interface série) et de la mémoire. Ces différents éléments sont reliés entre eux tel que présenté sur la Figure 20 :

[11] Think fait dans cette section référence à Think-Fractal mais nous conservons cette appellation afin de correspondre à la littérature actuelle.

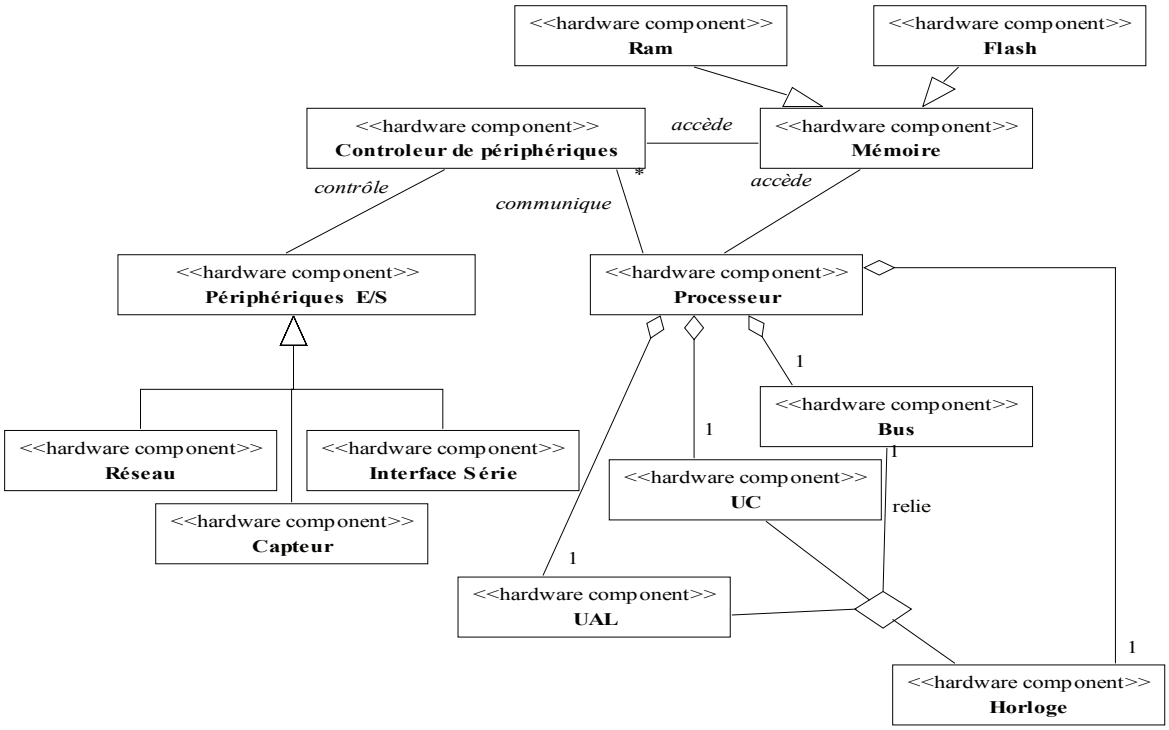


Figure 20 : "Componentization" d'un capteur

Le *processeur* est le centre du système. Il s'agit d'une unité capable d'interpréter la suite des instructions d'un programme puis de les exécuter séquentiellement. Pour cela, il est constitué d'une *Unité Arithmétique et Logique* (UAL) pour effectuer des opérations de calcul élémentaire (+, -) et de comparaison logique (<,>, =), d'une *Unité de Contrôle* (UC) pour utiliser les instructions stockées en mémoire, d'une *horloge* pour le cadencer et d'un *bus* pour relier ces différents composants. Les principaux processeurs utilisés dans les capteurs sont actuellement *AVR* et *ARM*. Les processeurs AVR [Wik06a] font partie de la famille des microprocesseurs RISC fabriqués par Atmel[12] et sont également basés sur une architecture Harvard qui stocke le programme et les données séparément. Les processeurs ARM [Wik06b] sont également basés sur une architecture RISC et sont largement utilisés dans les systèmes embarqués comme les PDAs.

En outre, les *périphériques d'Entrées-Sorties* (Périphériques d'E/S) sont les liens que possèdent le capteur avec l'environnement, les autres capteurs et éventuellement les utilisateurs. L'émetteur-récepteur (réseau), la connexion avec la machine de développement (interface Série) ou le capteur entrent dans cette catégorie. Chaque type de périphériques d'entrées-sorties possède son propre contrôleur appelé *contrôleur de périphériques*. Ce dernier est en charge de ces périphériques et les commande. Cela permet de les rendre autonomes. Le processeur est informé de la fin d'une opération d'entrée-sortie par l'interruption générée par le contrôleur.

Enfin, la *mémoire*, quant à elle, permet de stocker les données et les instructions nécessaires à l'exécution du programme. Le processeur tout comme les contrôleurs de périphériques y ont accès.

[12] http://www.atmel.com/

Dans Think, tous les éléments matériels sont directement réifiés en composants. Ces derniers fournissent exactement les mêmes fonctionnalités que le matériel sous-jacent, aucune abstraction supplémentaire n'est envisagée. Cela revient à remplacer les stéréotypes `<<hardware composant>>` de la Figure 20 par `<<software composant>>`. Ce diagramme n'est donc pas présenté ici.

Il faut maintenant déterminer la manière dont le système d'exploitation utilise ces composants pour réaliser les fonctionnalités de l'application. Ainsi, dans un système classique les différentes fonctionnalités sont représentées par des processus, appelés threads, qui sont alloués au processeur par un ordonnanceur. Ces threads utilisent des ressources. Cela peut se modéliser de la manière suivante :

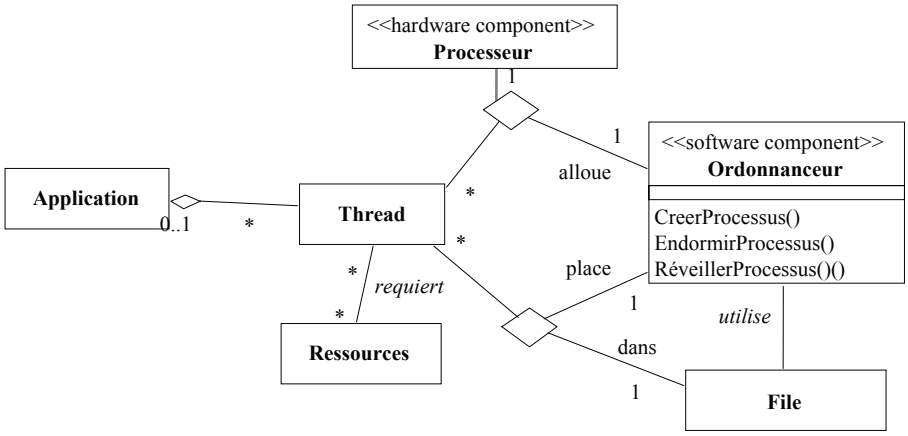


Figure 21 : Liaison entre le système d'exploitation et l'application

Or l'utilisation de threads pour le domaine des réseaux de capteurs semble être une solution assez coûteuse étant donné que chacun d'entre eux doit stocker dans la mémoire une copie du contexte d'exécution durant toute la durée de son existence. L'emploi d'un tel mécanisme consomme donc énormément de ressources mémoire et est par conséquent à envisager avec parcimonie.

De plus, même si différentes politiques d'ordonnancement peuvent être envisagées afin d'allouer un processus au processeur, il nous semble raisonnable d'envisager un ordonnancement via une file *FIFO*[13] comme c'est le cas dans TinyOS. En effet, il s'agit premièrement d'un algorithme simple à mettre en œuvre, deuxièmement le temps d'activité d'un mote devant être le plus court possible, les tâches de longue durée seront considérées comme marginales et troisièmement, comme nous sommes en présence d'un système « mono-utilisateur », si une tâche monopolise le processeur, il s'agit d'une situation acceptable. De plus, l'existence de mécanismes bloquant tel que l'attente de l'arrivée d'un message ne sera pas permis afin de ne pas empêcher l'exécution des autres processus. Ce choix est modélisé sur la Figure 21 au travers de l'élément `File`.

Quant à l'élément `Ressources` également présent sur cette Figure 21, il permet d'alléger le modèle car il représente indifféremment plusieurs éléments du modèle de la Figure 20. En effet, une ressource est un objet logiciel ou matériel pouvant être utilisé par un programme [Kra85]. La Figure 22 schématise cette définition dans le cadre des capteurs.

[13] FIFO abréviation couramment employée pour « First In, First Out » signifiant en français « Premier Arrivé, Premier Servi ».

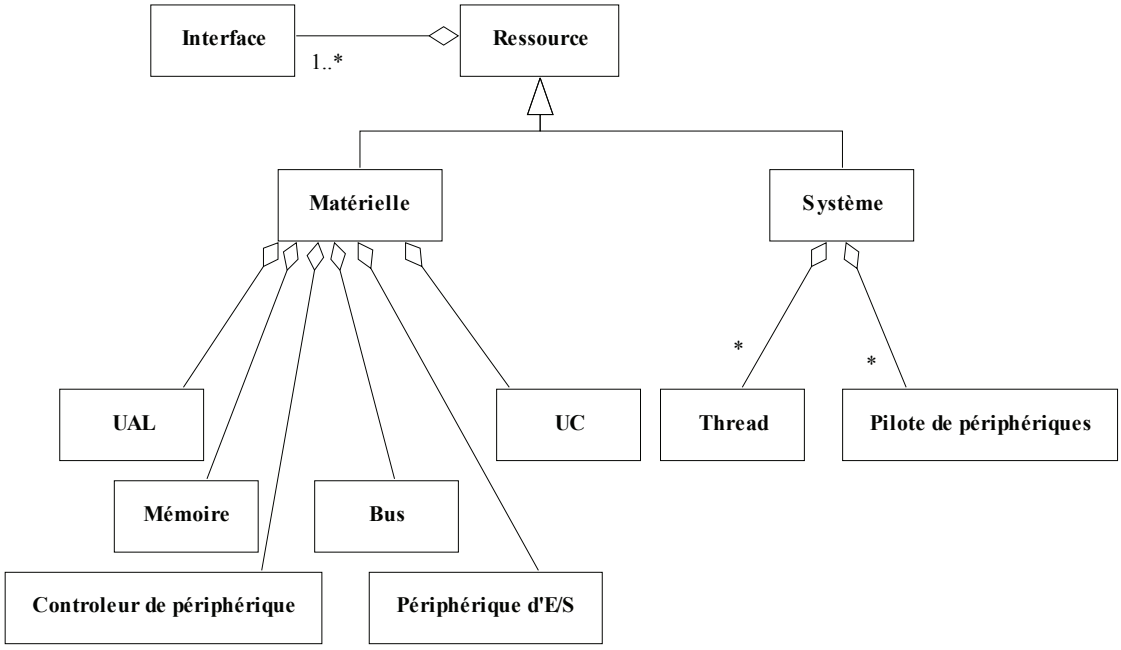


Figure 22 : Les ressources sur un capteur

Il ne reste maintenant plus qu'à déterminer le type de noyau du système d'exploitation s'adaptant le mieux à notre domaine d'étude. De façon évidente, un noyau monolithique est trop volumineux pour pouvoir être utilisé sur des capteurs. Un exonoyau semble bien adapté car ainsi l'application aura directement accès aux composants matériels dont elle aura besoin.

Nous avons décrit les différents éléments de base constituant notre modèle ainsi que le type de noyau pouvant correspondre au domaine des réseaux de capteurs. Il ne nous reste plus qu'à déterminer la manière dont ces différents éléments interagiront entre eux. Un modèle basé composant repose classiquement sur le modèle client-serveur. En effet, une interface cliente demande l'exécution d'une requête, i.e. d'une fonction, sur une interface serveur d'un autre composant mais un modèle de communication par message peut également être utilisé. Think n'impose rien concernant la nature des communications.

Or la nature des réseaux de capteurs est fortement basée évènement du fait premièrement des fortes contraintes existantes et deuxièmement du lien très étroit entre le capteur et son environnement proche. Ce sera en effet souvent le capteur qui sera à l'origine de la détection d'un phénomène et qui en avertira le système. Ce phénomène peut être mis en évidence en étudiant les différents états que peut prendre un capteur. Ceux-ci sont au nombre de quatre : l'état `Init`, qui lui permet de s'initialiser ; l'état `Activité`, qui se produit lorsqu'il réalise une tâche ; l'état `Idle`, qui lui permet d'économiser de l'énergie en se mettant en attente d'évènement et l'état `Sleep` qui permet d'arrêter tous les composants. Dans le diagramme d'états suivants, l'état `Idle` correspond aux états `AttenteEvt`, `AttenteVoisinage` et l'état `Sleep` n'est pas représenté. Comme on peut le constater sur le schéma Figure 23, un évènement est toujours à l'origine de l'activité du capteur.

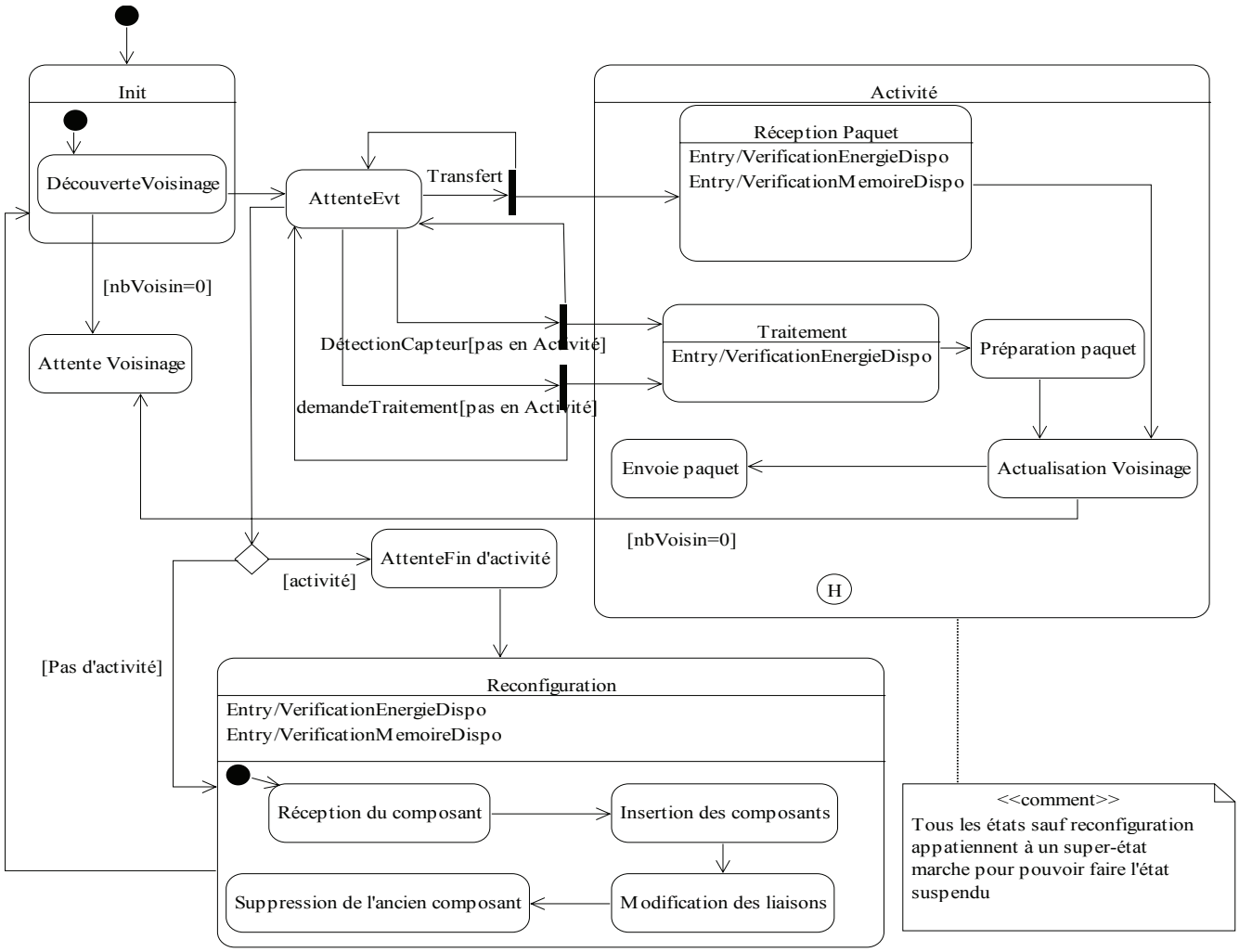


Figure 23 : Les états d'un capteur

Ce diagramme d'états (Figure 23) met en évidence la coexistence de deux états : `Idle` et `Activité`. En effet, lors de la réalisation d'un traitement, comme l'agrégation des données, un évènement peut se produire. Par exemple, la batterie peut signaler qu'elle est trop faible pour continuer à alimenter le capteur. Il convient donc d'avoir un mécanisme permettant de récupérer ces évènements pour pouvoir les traiter. Cela implique donc deux « threads » afin de gérer premièrement les attentes d'évènement et deuxièmement les activités en cours dans le capteur. Cela pose donc un deuxième problème : comment faire coexister l'aspect évènementiel avec l'aspect fonctionnel des composants (appels de fonctions notamment) ?

Une première approche pourrait être de générer un modèle d'exécution complètement évènementiel c'est-à-dire que tous les échanges du système se feraient désormais sous forme d'évènements et non plus d'appel de fonctions. Un inconvénient de cette méthode est qu'il faut déterminer le « chemin » suivi par les évènements c'est-à-dire qu'il faut déterminer quel est le composant qui génère un évènement et quels sont ceux qui le traitent. Par conséquent, une modification du système peut se révéler compliquée.

Une deuxième approche pourrait être d'utiliser le concept des MessageDrivenBean existant en EJB et de les appliquer aux évènements. Ainsi, tous les composants susceptibles de signaler un évènement publient un message dans la file d'attente des évènements. Le composant gérant l'ordonnancement s'inscrit comme abonnés de cette file. Les messages présents dans la file d'évènements sont prioritaires sur les tâches présentes dans l'ordonnanceur. Ainsi lorsqu'un évènement se produit, il est placé dans la file d'attente des évènements, l'ordonnanceur en extrait la fonction associée et la déclenche. Tous les appels de fonctions concernant cet évènement seront dans la file d'attente de l'ordonnanceur.

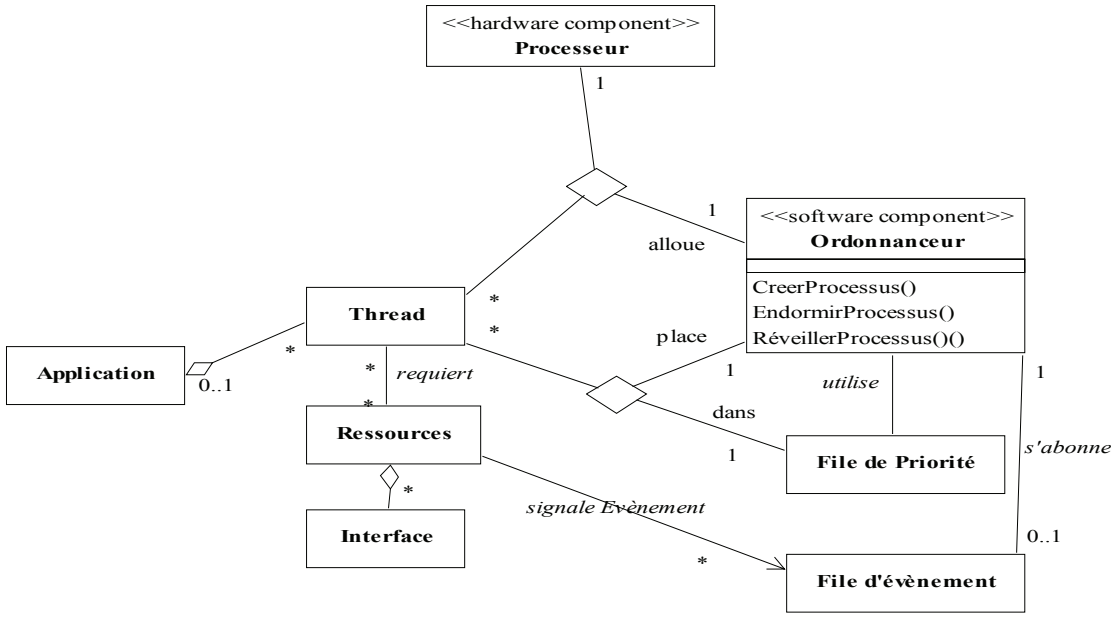


Figure 24 : Modélisation de l'utilisation d'une file d'évènement

Cela montre la nécessité de définir premièrement un nouvel ordonnanceur qui tienne compte de la file d'attente des évènements et qui soit capable d'extraire la fonction à appeler de cet évènement. Il convient donc de déterminer des « patterns » d'évènements afin de généraliser ce mécanisme. Malheureusement, si une partie de l'application requiert l'arrivée d'un évènement pour fonctionner, elle n'en sera pas informée. Cette solution ne convient donc pas car dans ce cas, il conviendrait alors d'avoir une file pour chaque type d'évènement pouvant être déclenché afin que les composants puissent s'y abonner. Ce mécanisme engendrerait donc un surcoût mémoire inacceptable pour les capteurs.

Une dernière possibilité consiste donc à ce que chaque composant requérant la présence d'un évènement pour pouvoir fonctionner possède un « `gestionnaire d'évènement` » correspondant à l'évènement dont il a besoin. Et réciproquement, que chaque composant pouvant signaler un évènement puisse le faire. Cela implique d'avoir une nouvelle structure de composant. Celle-ci est présentée sur la Figure 25 suivante :

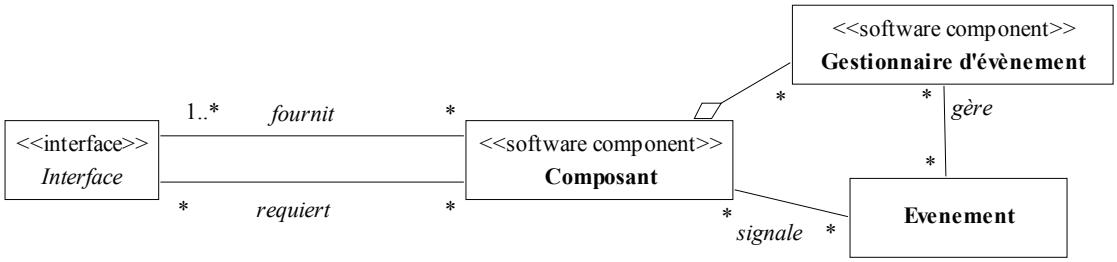


Figure 25 : Modèle de composant basé évènement

L'implémentation du « `gestionnaire d'évènements` » en tant que sous-composant pourrait faciliter les modifications ultérieures à apporter au système. Ainsi si de nouveaux évènements doivent être pris en compte ou s'il est nécessaire de modifier des évènements existants, seul le gestionnaire devra être modifié et éventuellement les liens entre les évènements écoutés et le gestionnaire.

Finalement, notre système d'exploitation repose sur un modèle basé évènement fonctionnant de la manière suivante : les composants de plus bas niveau signalent des évènements aux composants de niveau supérieurs qui sont capables de les traiter et les composants de niveaux supérieurs peuvent demander l'exécution de tâches à des composants de niveaux inférieurs via des appels de fonction. De même, des composants d'un même niveau communiquent également par des appels de fonctions. La Figure 26 schématise se mécanisme.

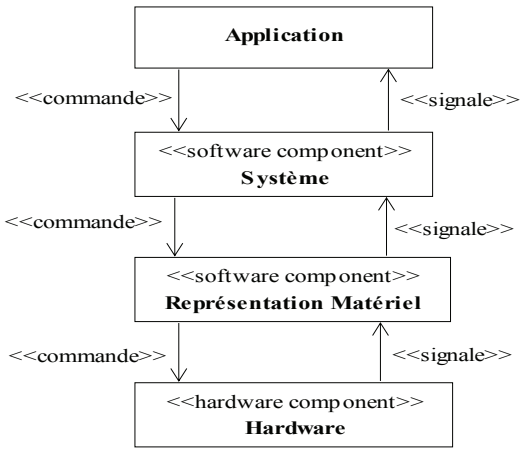


Figure 26 : Fonctionnement du système

L'ordonnanceur est en charge du mécanisme de la répartition du processeur entre les différents éléments. Or, le processus d'ordonnancement choisi précédemment, une file FIFO, impose que toutes les tâches soient traitées dans l'ordre d'arrivée. Ainsi, l'arrivée d'un évènement n'implique pas forcément la prise en compte immédiate de celui-ci. Une solution à ce problème réside dans l'utilisation d'une « `file de priorité` » dans laquelle les évènements auront une priorité supérieure aux appels de fonctions. Dans ce cas, lors d'une arrivée d'un évènement, l'ordonnanceur placera ce dernier en tête de la file, ou s'il existe d'autres tâches ayant la même priorité à la suite de ces dernières. Une fois, le traitement en cours terminé, l'évènement sera pris en compte et l'ordonnanceur le signalera à tous les composants possédant le « gestionnaire d'évènement » adéquat qui est capable de le traiter.

Dans cette section, nous avons présenté les limites du modèle TinyOS pour des applications envisageables dans un futur proche. Ainsi le besoin de reconfiguration dynamique et par extension l'adaptativité ne sont actuellement pas des fonctionnalités envisagées pour les applications basées sur les réseaux de capteurs. Nous avons donc proposé des concepts permettant de définir un système d'exploitation les prenant en compte. Pour cela, nous avons tenté de marier les aspects éprouvés de TinyOS notamment le modèle basé évènement avec les possibilités du modèle Think et en particulier les aspects dynamiques offerts. Dans la section suivante, nous allons concevoir un mécanisme de reconfiguration dynamique applicable aux réseaux de capteurs et au système d'exploitation que nous venons de définir.

IV. Etude de la reconfiguration dynamique

La *reconfiguration dynamique* est une opération permettant de remplacer un composant par un autre dans une application en cours d'exécution. Plusieurs causes peuvent être à l'origine d'une telle action. Il peut par exemple être nécessaire de substituer un composant mal implémenté, c'est-à-dire ne réalisant les fonctionnalités prévues ou les réalisant de manière incorrecte, ou encore d'ajouter à ce composant de nouvelles fonctionnalités. Un tel processus ne peut être réalisé qu'à la seule condition que le composant soit dans un *état stable*, i.e. qu'il ne soit plus utilisé. Dans le cas contraire, la reconfiguration pourrait conduire à un crash irrémédiable du système.

Cette section propose ainsi une modélisation d'un mécanisme de reconfiguration dynamique adapté aux réseaux de capteurs. Nous commençons dans un premier temps par décrire le mécanisme générique proposé dans [Pol04] car celui-ci repose déjà sur le modèle Think-Fractal mais n'est pas assez contraint pour être directement adaptable aux réseaux de capteurs. Nous proposons donc dans un deuxième temps les modifications à apporter à ce mécanisme pour le rendre applicable pour ce domaine.

IV.1) Présentation de la solution proposée par Juraj Polakovic

Au cours de son stage de master [Pol04], Juraj Polakovic a proposé un mécanisme de reconfiguration dynamique pour Think reposant sur la notion de contrôleurs telle que présentée dans le modèle Fractal. La reconfiguration est ici envisagée au niveau des composants et les objectifs recherchés pour ce mécanisme sont la généricité et la flexibilité pour supporter tous les modèles de reconfiguration dynamique, la non-modification de l'aspect fonctionnel des composants et la minimisation de la surcharge mémoire à l'exécution. De plus, seuls les composants pouvant nécessiter d'être reconfigurés possèdent ce mécanisme : il est donc considéré comme optionnel.

Afin de réaliser ces objectifs, la reconfiguration repose dans ce modèle sur une description de la reconfiguration dynamique constituée premièrement par l'architecture du système et deuxièmement par un ensemble de règles. Celles-ci définissent si la reconfiguration est réalisable ou non ainsi que la succession des étapes à suivre pour la substitution d'un composant par un autre. Ces deux fonctionnalités sont implémentées via un outil qui contiendra donc une représentation de l'architecture ainsi que la marche à suivre pour la reconfiguration. Il s'agit en réalité d'un nouveau contrôleur, appelé `ReconfEngine`, qui est relié aux contrôleurs Fractal déjà existant. La Figure 27 présente les liens entre le `ReconfEngine` et les autres contrôleurs du système. Le système à reconfigurer peut ainsi être soit un composant « primitif », soit un composant « composite » ou soit une application.

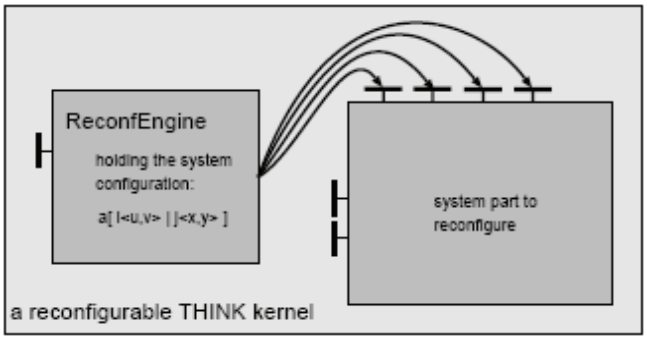


Figure 27 : Liaison entre le contrôleur de reconfiguration et le système à reconfigurer [P04]

En conséquence pour implémenter ce nouveau contrôleur, une nouvelle interface a été définie. Celle-ci décrit trois méthodes : une méthode `request()` permettant à l'initiateur de la reconfiguration de soumettre une nouvelle description de la configuration du système ; une méthode `resume()` pour éviter de rester bloquer lors de l'attente d'un état stable et une methode `cancel()` pour annuler une reconfiguration et revenir dans la configuration initiale. Cela implique donc que le contrôleur de reconfiguration stocke la configuration courante du système. On obtient alors la description d'interface suivante :

```
interface Reconfiguration {
   int request(string description) ;
   void resume() ;
   int cancel() ;
}
```

En outre, le contrôleur `LifeCycleController` de Fractal a été étendu avec deux nouvelles méthodes : une première méthode `suspend()` dont l'objectif est de placer le composant dans un état stable et si besoin est ses sous-composants. La deuxième méthode `resume()` qui consiste à annuler la reconfiguration en cours et à relancer l'activité normale du composant et éventuellement de ses sous-composants.

Ainsi, le mécanisme proposé fonctionne de la manière suivante :

1. Un initiateur demande une reconfiguration en appelant la fonction `request()` du `ReconfEngine` en passant la nouvelle configuration comme argument
2. Le `ReconfEngine` vérifie ensuite si la reconfiguration peut être réalisée. Dans le cas contraire, il la rejette et s'arrête. A cette étape, les composants et les liaisons devant être reconfigurés sont connus.
3. Le `ReconfEngine` se lie alors aux interfaces de contrôle des composants à reconfigurer c'est-à-dire le `LifeCycleController`, le `ContentController`, `BindingController`, …
4. Puis pour chaque composant à reconfigurer, le `RequestEngine` demande l'obtention d'un état stable en appelant la méthode `suspend()` de l'extension `LifeCycleController`.
5. Quand l'état stable est atteint, le `LifeCycleController` réactive le `ReconfEngine` en utilisant la fonction `resume()`
6. Le `ReconfEngine` met finalement en place la nouvelle configuration :
    a. Les nouveaux composants sont chargés en mémoire

b. Et/ou les liaisons sont reconfigurées (`BindingController`, `ContentController`)
c. Les états sont transférés entre les composants (`AttributeController`)
d. Les composants qui ne sont plus utilisés sont déchargés de la mémoire

7. La reconfiguration est ainsi terminée et la nouvelle configuration est sauvegardée.

Enfin dans [Pol04], deux solutions ont été proposées pour déterminer l'état stable d'un composant. La première reproduit un mécanisme utilisé dans Julia pour effectuer cette tâche. Il s'agit du mécanisme d'interception. Celui-ci consiste à interposer des *proxis* entre les interfaces externes et internes d'un composant afin de comptabiliser à l'aide d'un compteur les appels entrants et sortants de ce composant. Lorsque le compteur est à zéro, il n'y a plus de fonction en cours dans le composant, il n'y a donc plus d'activité. Le composant est dans un état stable. Cette approche pose des problèmes de performance. En effet, chaque *proxi* occupe de l'espace mémoire supplémentaire. Il empêche également l'utilisation des techniques de *by-pass* qui permettent d'appeler directement une interface d'un sous-composant grâce à une référence vers cette dernière. Sans cette optimisation, l'appel d'une méthode d'un sous-composant requiert l'utilisation de trois indirections.

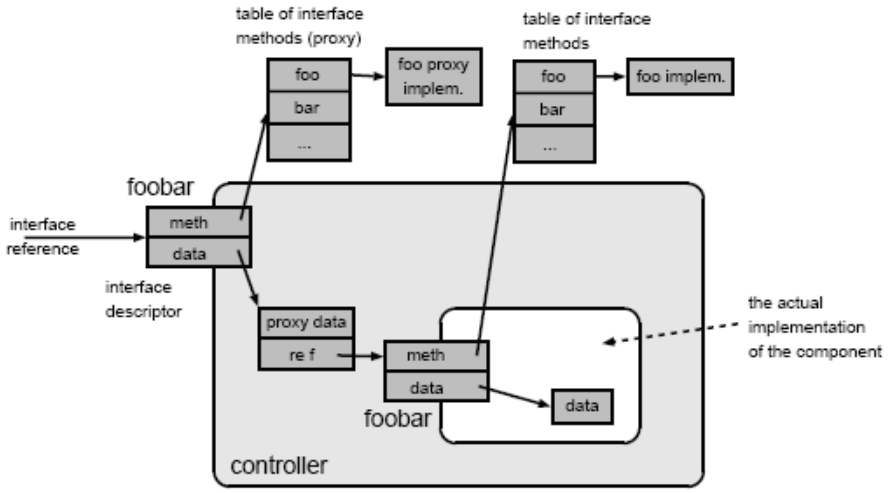


Figure 28 : Implémentation des *proxis* d'un composant [Pol04]

La deuxième solution, basée sur le mécanisme K42, ne sera pas détaillée ici car elle n'est pas adaptée au contexte des réseaux de capteurs. En effet, celle-ci repose sur l'utilisation d'un composant de gestion de génération de threads et suppose également que ces threads se terminent toujours dans un temps limité et sont non-bloquants. Or il s'agit là d'un mécanisme trop gourmand en ressource mémoire pour être utilisé dans le cadre de notre étude car chaque thread possède une copie locale du contexte d'exécution.

IV.2) Les améliorations du modèle

Le modèle de mécanisme de reconfiguration dynamique proposé pour Think [Pol04] est trop générique pour satisfaire les contraintes inhérentes au domaine des réseaux de capteurs. Une modification de cette proposition est donc nécessaire.

Premièrement, nous n'utilisons pas un modèle « classique » de système d'exploitation mais le modèle d'exécution basé évènement que nous avons décrit précédemment. En effet, ce modèle reprend non seulement une grande partie des caractéristiques qui ont fait le succès de TinyOS mais autorise de plus les allocations dynamiques de composants permettant ainsi d'envisager des mécanismes avancés telle que notamment la reconfiguration dynamique. Pour mémoire, le système fonctionne de la manière telle que modélisée sur la Figure 29 :

1. Initialisation du système.
2. Attente d'évènement.
3. L'arrivée d'un évènement provoque une activité. Le système se maintient néanmoins en attente d'évènement.
4. Le système peut être arrêté par une intervention de l'utilisateur et si l'utilisateur le réactive, il reprend dans l'état où il s'était arrêté.

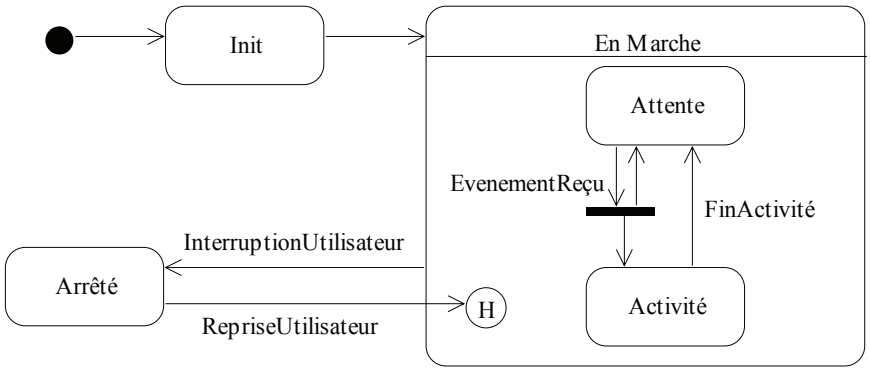


Figure 29 : Les états du système

En nous appuyant sur ce modèle évènementiel, on s'aperçoit rapidement que la recherche d'un état stable est une opération inutile voire coûteuse. En effet, dans notre système à un instant donné, une seule tâche est en cours d'exécution. Cela implique que lorsque cette dernière se termine, le système est stable et après soit il exécute la prochaine tâche présente dans l'ordonnanceur, soit il se remet en attente d'évènement. De plus, si on considère la reconfiguration comme un évènement, il est possible de lui attribuer une priorité élevée par rapport aux autres tâches et ainsi permettre que la prochaine tâche sélectionnée par l'ordonnanceur soit la reconfiguration. La suppression de l'état stable est en outre rendue possible par le mode de fonctionnement des capteurs. En effet, il est préconisé que le capteur exécute sa tâche le plus rapidement possible et se remette en attente afin d'économiser la consommation des ressources. Ainsi, un capteur doit passer le plus clair de son temps en attente. En outre, une contrainte de notre modèle est d'interdire les fonctions bloquantes. On peut donc considérer que si l'évènement reconfiguration survient pendant l'exécution d'une tâche, celle-ci mettra un temps relativement court pour se terminer et la reconfiguration pourra avoir lieu.

Le coût de l'attente de fin d'activité devra faire l'objet d'une étude où il devra être comparé au coût de la recherche d'un état stable pour chaque composant. Des éléments peuvent néanmoins être apportés pour prouver l'intérêt de cette solution. Premièrement, les techniques de by-pass, qui constituent une des optimisations utilisées par Think, peuvent être conservées. Deuxièmement, il n'est plus nécessaire d'interposer des *proxis* pour comptabiliser les appels entrants et sortants d'un composant et l'extension du `LifeCycleController` peut également être supprimée. Cela permet ainsi d'économiser de l'espace mémoire par rapport à la solution envisagée par Polakovic.

La deuxième modification que nous pouvons envisager concerne les tests déterminant si la reconfiguration peut être réalisée. Compte tenu du faible espace mémoire disponible, il ne nous parait pas indiqué de conserver ce mécanisme en l'état sur le capteur car il nécessite de stocker non seulement la description de l'architecture mais également les règles autorisant la transformation de cette architecture. Il est cependant possible de ne stocker que les règles permettant les transformations. En effet, grâce au modèle Fractal, des moyens d'introspection du composant ont été introduits via les contrôleurs notamment le `ContentController`, le `BindingController` et le `Component`. Ceux-ci permettent d'obtenir la structure globale mais ce mécanisme augmente l'utilisation du processeur. Le temps d'établissement de l'architecture sera inévitablement fonction de la complexité architecturelle du capteur comme lorsque on est par exemple en présence de sous-composants.

Nous préconisons donc une délocalisation de l'ensemble du mécanisme. Un collecteur, possédant des contraintes moins sévères, ou une station de base pourraient réaliser cette fonctionnalité. Ainsi dans le cadre d'une reconfiguration, la station de base par exemple fonctionnerait de la manière suivante :

1. Localement, elle possède la description de l'architecture du capteur ainsi que les règles de reconfiguration applicable pour celui-ci. Ce qui lui permet :
    a. premièrement de tester si la reconfiguration est faisable
    b. et deuxièmement d'établir la procédure à suivre pour modifier le capteur.
2. Une fois, la procédure de reconfiguration établie (quelle(s) liaison(s) modifiée(s), quel(s) composant(s) à enlever, quel(s) composant(s) à ajouter, etc.), la station de base vérifie si le ou les composant(s) à ajouter sont déjà présents sur le capteur.
    a. Si c'est le cas, elle transmet seulement au `ReconfEngine` la demande de reconfiguration avec la marche à suivre.
    b. Sinon, elle transmet en plus le nouveau composant ou bien, si un capteur plus proche le possède, la localisation de celui-ci afin d'en obtenir une copie.

Il existe néanmoins deux inconvénients majeurs à cette proposition. Le premier concerne l'augmentation des messages échangés sur le réseau. En effet, si l'initiateur de la reconfiguration est par exemple un capteur, la demande de reconfiguration doit être transmise à la station de base pour être traitée, puis la reconfiguration est transmise de la station de base au capteur concerné et une fois, celle-ci réalisée le capteur doit informer la station de base que l'opération s'est correctement déroulée. Cet exemple met en évidence l'augmentation des messages échangés sur le réseau. Il faut au minimum trois messages alors que la solution de Polakovic n'en nécessite qu'un : l'initialisation de la reconfiguration.

Le deuxième inconvénient concerne l'aspect centralisé du mécanisme. En effet, intuitivement si la station de base tombe en panne, il n'est plus possible de reconfigurer le système. Mais il ne s'agit pas là de l'exemple le plus handicapant. Un problème beaucoup plus gênant apparaît si par exemple, un message de validation de reconfiguration se perd. La station de base peut alors considérer que la reconfiguration a échoué et alors soit elle considère le capteur comme détruit, soit elle considère qu'il est revenu dans sa configuration initiale alors que ce dernier a bien été modifié. Ce cas peut effectivement se produire car la fiabilité des échanges n'est pas une condition imposée dans les réseaux de capteurs.

Il conviendra donc d'évaluer cette proposition et le taux de pertes de messages dans les réseaux de capteurs à trois moments clés : au début de la vie du réseau, en milieu et en fin de vie. De plus, le mécanisme de test proposé par Polakovic n'ayant pas vraiment été réalisé, une implémentation de ce mécanisme devra être faîte afin de premièrement déterminer

l'espace mémoire occupé par celui-ci ; deuxièmement, évaluer l'espace mémoire et le taux d'occupation du processeur si on supprime la description de l'architecture locale et troisièmement effectuer une comparaison avec le mécanisme délocalisé. Cela permettra ainsi de déterminer s'il vaut mieux sacrifier de l'espace mémoire ou le trafic dans le réseau.

Pour en terminer avec cette proposition, nous présentons ici le scénario d'utilisation envisagée lors d'une reconfiguration dynamique. Nous détaillons ici uniquement les étapes concernant le seul capteur à reconfigurer, les étapes pour l'initiateur ayant été détaillées pour la station de base.

1. Réception d'un évènement correspondant à une reconfiguration.
2. Traitement de l'évènement par l'ordonnanceur et le (ou les) gestionnaire(s) correspondant(s) :
    a. Suspension de la tâche en cours par l'ordonnanceur
    b. Placement de la tâche de reconfiguration en tête de la file de priorité par l'ordonnanceur
    c. Placement des éléments de reconfiguration aux endroits adéquats (composants dans la mémoire, règles de reconfiguration dans le `ReconfEngine`)
    d. Réactivation de la tâche suspendue
3. Exécution de la reconfiguration
    a. Vérification de l'énergie disponible et de l'espace mémoire. Si les ressources sont insuffisantes, un message d'échec est renvoyé à l'initiateur. Le composant reste dans sa configuration initiale.
    b. Le composant `ReconfEngine` est lié aux différents contrôleurs.
    c. Le composant `ReconfEngine` réalise alors les traitements suivants :
        i. Chargement des nouveaux composants en mémoire permanente
        ii. Reconfiguration des liaisons via le `BindingController` et le `ContentController`
        iii. Transfert des états entre les composants si besoin est via l'`AttributeController`.
        iv. Suppression de la mémoire permanente des anciens composants, i.e. des composants désormais inutilisés.
    d. Information du succès de la reconfiguration via l'envoi d'un message à l'initiateur.

Dans cette section, nous avons établi un nouveau mécanisme de reconfiguration dynamique adapté aux contraintes inhérentes aux réseaux de capteurs en nous aidant d'une proposition générique déjà réalisée pour Think et du système d'exploitation défini à la section III. Cependant cette proposition est incomplète, il manque en effet une implémentation ainsi que la mise en place de procédures de tests. La section suivante montre les difficultés liées à l'utilisation de Think dans la phase d'implémentation.

V. Les limites de Think pour implémenter le modèle proposé

L'absence de composants pour AVR dans la bibliothèque Kortex doit être signalée comme la première limite à l'utilisation de Think dans le domaine des réseaux de capteurs. Les microprocesseurs AVR sont en effet ceux utilisés sur la plupart des motes fournis avec TinyOS. Or ces composants devant être en accord avec le matériel sous-jacent, il nécessite une implémentation particulière ; certains fichiers devant notamment être écrits en assembleur. Cependant ces composants existent déjà mais leur utilisation, même à des fins de recherche, est subordonnée à un accord préalable avec France Télécom R&D que nous n'avons pas obtenu.

Il a néanmoins été possible de tester certaines possibilités de Think mais uniquement pour les capteurs fournis par Intel c'est-à-dire les « iMotes ». En effet, ces derniers possèdent un microcontrôleur ARM dont une partie des composants est fourni dans la version 3 de Think. Malheureusement, il s'agit là d'une version en cours de développement. La documentation associée à cette version étant très incomplète, nous avons dû nous contenter de tester les exemples fournis afin de déterminer si Think pouvait s'adapter aux contraintes des réseaux de capteurs. Hormis un exemple où la taille de l'image binaire générée était supérieure à la capacité mémoire des iMotes, tous les autres exemples étaient compatibles. Cela démontre la possibilité d'utiliser un système reposant sur Think pour des capteurs à condition toutefois de ne pas oublier d'optimiser les codes réalisés. En outre, un émulateur a été installé afin de tester l'exécution de ces programmes. La procédure d'installation et d'utilisation de cet émulateur est détaillée en annexes.

D'une manière plus générale, un obstacle fondamental à l'utilisation de Think est un manque de documentation. Le didacticiel [Pul] en est un exemple flagrant : de nombreuses rubriques par exemple ne sont pas renseignées. De plus, certaines sections contiennent des informations erronées ou non valides. Par exemple, l'héritage d'interfaces est indiqué comme réalisable dans la documentation[14] fournie avec la version 3. Or dans la pratique, son utilisation s'avère impossible sauf à la condition de décrire à nouveau dans l'interface fille toutes les interfaces présentes dans l'interface mère. Cette non-implémentation a été mise en évidence via l'application également détaillée dans Annexe E. Cependant, la création récente d'un espace de documentation présage de la volonté des concepteurs de Think d'améliorer le contenu de leur documentation.

Enfin, la procédure d'installation est également une limitation à l'utilisation de Think car il convient en premier lieu de connaître le processeur cible. En effet, l'installation d'un compilateur croisé est requis et cela se révèle être une tâche particulièrement ardue. En effet, un *compilateur croisé* est utilisé lorsqu'on possède une machine de développement et que l'application développée sur cette machine sera en réalité installée et exécutée sur une autre, qui ne possède pas forcément les mêmes caractéristiques. Ainsi, dans notre cas, la machine de développement était une machine i686 sous *Ubuntu Breeze* et le processeur cible du capteur un ARM. Le premier compilateur croisé pour ARM que nous avons installé[15] ne permettait pas de gérer les « float ». Or, cela est possible à condition de trouver la version de gcc compatible non seulement avec la version de Arm supportant les float mais également avec la bonne version de la librairie libc. Nous avons tenté différentes possibilités mais sans obtenir de succès. Il est à noter que chaque tentative d'installation de ce compilateur croisé prenait en moyenne trois heures.

[14] la partie de la documentation concernant les interfaces est fournit en annexe,
[15] c'est l'installation de ce compilateur croisé qui est détaillé en annexes

VI. Conclusion

Au cours de ce stage, nous avons cherché à déterminer si l'utilisation des principes et des apports d'une approche basée véritablement sur l'ingénierie logicielle basée composant pouvait être conciliable avec les réseaux de capteurs. Afin de répondre à cette problématique, nous avons, dans un premier temps, effectué des recherches sur le domaine des réseaux de capteurs afin d'identifier les contraintes associées aux capteurs et à leurs mises en réseau. Une fois, ces principes de base fixés, nous nous sommes, dans un deuxième temps, concentrés sur l'étude du système d'exploitation de référence à savoir TinyOS. Cela nous a permis de dégager non seulement des mécanismes éprouvés mais également les limites de ce système. Une limite fondamentale a notamment été mise en évidence, il s'agit de l'absence de mécanisme de reconfiguration dynamique. Parallèlement à cette analyse, nous avons également exploré Think, l'implémentation de Fractal pour les systèmes embarqués. Il s'est avéré que Think est en réalité un modèle permettant de créer non seulement des systèmes d'exploitation mais également les applications fonctionnant dessus. En outre, ce modèle propose les aspects dynamiques manquant à TinyOS. En nous appuyant sur une comparaison des avantages et des inconvénients de ces deux modèles, nous avons alors dégagé les éléments fondamentaux nécessaires à la conception d'un système d'exploitation reconfigurable dynamiquement pour les réseaux de capteurs ainsi qu'un mécanisme de reconfiguration pour ce système d'exploitation.

La difficulté majeure à laquelle nous avons été confrontés lors de ce travail réside dans l'installation et la configuration de Think. En effet, l'installation d'un compilateur croisé est premièrement une opération non triviale et deuxièmement, la création de la chaîne de construction de Think nécessite de modifier différents fichiers – dont les trois fichiers de configuration et les outils de construction programmés en Java. Cette modification est nécessaire afin de rendre cette chaîne de construction fonctionnelle sur la machine de développement. Une autre difficulté majeure a consisté à devoir travailler sur une version en cours de développement et donc nécessitant de nombreux débuggages afin de pouvoir compiler, même, les exemples fournis. La gestion des contraintes propres aux capteurs, la compréhension du fonctionnement des modèles TinyOS et Think ainsi que le manque de documentation existante sur Think notamment pour ce qui concerne la partie technique ont également présenté quelques difficultés.

Une modélisation du système d'exploitation reconfigurable et des mécanismes de reconfiguration ayant été proposés, une perspective essentielle à ce travail serait de réaliser les implémentations correspondantes. Une fois celles-ci concrêtisées, des tests d'évaluation pourraient ainsi être effectués afin d'éprouver le respect des contraintes inhérentes aux capteurs notamment l'espace mémoire occupé. Cependant, il conviendra d'attendre qu'une version stable de la version 3 de Think-Fractal soit effectivement développée mais également que les composants pour AVR soient disponibles afin que ces tests soient aussi appliqués au processeur le plus répandu sur les motes.

Bibliographie

[BCS02]	Eric Bruneton, Thierry Coupaye and Jean-Bernard Stefani. *Recursive and Dynamic Software Composition with Sharing.* In Seventh International Workshop on Component-Oriented Programming (WCOP02), Malaga, Spain, June 10-14, 2002.
[BCS04]	Eric Bruneton, Thierry Coupaye and Jean-Bernard Stefani. *The Fractal Component Model Specification*, ObjectWeb Consortium, February 5, 2004, Draft, version 2.0-3. http://fractal.objectweb.org.
[BHRT04]	Jan Blumenthal, Matthias Handy, Frank Reichenbach, Dirk Timmermann. *SeNeTs - Test and Validation Environment for Applications in Large-Scale Wireless Sensor Networks*. In *2nd IEEE International Conference on Industrial Informatics, 2004*.
[Bru03]	Eric Bruneton, *Le modèle de composants Fractal*, ICAR'03 Ecole d'été sur les Intergiciels et la Construction d'Application et Reparties, 2003
[CES04]	David Culler, Deborah Estrin and Mani Srivastava. *Overview of Sensor Networks*. In IEEE Computer, vol. 37, no. 8, pp 41–49, august 2004.
[Fas01]	Jean-Philippe Fassino. *THINK : vers une architecture de systèmes flexibles*. Thèse de doctorat. 11 décembre 2001.
[FSLM02]	Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, Gilles Muller. *THINK A Software Framework for Component-based Operating System Kernels*. In the USENIX Annual Technical Conference, pages 73--86, Monterey, CA, USA, June 2002.
[GLBWBC03]	David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer and David Culler. *The nesC Language: A Holistic Approach to Networked Embedded Systems*. In Proceedings of Programming Language Design and Implementation (PLDI), June 2003.
[GLCB03]	David Gay, Philip Levis, David Culler and Eric Brewer. *nesC 1.1 Language Reference Manual*. In TinyOS documentation site, May 2003. http://nescc.sourceforge.net/papers/nesc-ref.pdf
[HSWHCP00a]	Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler and Kristofer Pister. *System architecture directions for network sensors*. In Proceedings of Ninth International Conference ASPLOS, Cambridge, MA, USA, November 2000.
[HSWHCP00b]	Jason Hill and Robert Szewczyk and Alec Woo and Seth Hollar and David Culler and Kristofer Pister. *System architecture directions for network sensors*. In ASPLOS, Cambridge, MA, November 2000.

[Hub00]	Michel HUBIN. *Traité sur les capteurs et la conception instrumentale.* 2000. http://perso.wanadoo.fr/michel.hubin/capteurs/instru.htm
[IEEE-1451 Expo2001]	*A Standard Smart Transducer Interface*, Sensors Expo, Philadelphia, Oct. 2001.
[Kra85]	Sacha Krakowiak *Principe des systèmes d'exploitation des ordinateurs*. Dunod Informatique, Paris, 1985
[KC03]	Jeffrey Kephart and David Chess. *The vision of Autonomic Computing. In IEEE Computer*. 36(1), pp. 41-50, 2003
[KDM05]	I. Khemapech, I. Duncan and A. Miller. *A survey of wireless sensor networks technology*. In PGNET, Proceedings of the 6th Annual PostGraduate Symposium on the Convergence of Telecommunications, Networking & Broadcasting, June 2005.
[KNEKLM04]	Sachin Kogekar, Sandeep Neema, Brandon Eames, Xenofon Kousoukos, Akos Ledeczi and Miklos Maroti. *Constraint-Guided Dynamic Reconfiguration in Sensor Networks*. In Proc. of Information Processing in Sensor Networks, IPSN '04, Berkeley, California, April 26-27, 2004.
[Lew04]	Franck L. Lewis. Wireless Sensor Networks. In Smart Environments : Technology, Protocols and Applications, ed. Diane Cook and Sajal Das, John Willey, New York, 2004.
[Liu05]	Ke Liu. *TinyOS/Motes, nesC Tutorial*. Présentation. Departement of Computer Science, SUNY Binghamton. Spring, 2005.
[LM&all04]	Konrad Lorincz, David J. Malan, Thaddeus R.F. Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoffrey Mainland, Matt Welsh, Steve Moulton, "Sensor Networks for Emergency Response: Challenges and Opportunities," *IEEE Pervasive Computing*, vol. 03, no. 4, pp. 16-23, Oct-Dec, 2004.
[LMGPSWBC04]	P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "*The emergence of networking abstractions and techniques in tinyOS*," in First Symposium on networked system design and implementation (NSDI04), San Francisco, California, USA, 2004, pp. 1--14.
[Lob]	Olivier Lobry. *THINK A Software Framework for Component-based Operating System Kernels*. Brochure. January 2006. http://www.objectweb.org/wws/d_read/marketing/public/projects/Think.pdf
[Mai06]	Laetitia Mailhes. Une nouvelle génération de mémoires magnétiques. Dans Les Echos, no. 19611, Innovation, pp. 30, 22 février 2006.

[Mau]	William Maurer. *The Scientist and Engineer's Guide to TinyOS Programming, Chapter 3 : TinyOS Components Based OS.* http://ttdp.org/tpg/html/book/c525.htm
[Pol04]	Juraj Polakovic. *Dynamische Rekonfiguration in THINK*. Rapport de master. Universität Karlsruhe (TH), 30 Juni 2004.
[Pul]	Jacques Pulou. THINK Tutorial - Getting Started with Fractal Think. http://think.objectweb.org/pdf/tutorial_think.pdf
[SGM02]	Clemens Szyperski, Dominik Gruntz and Stephan Murer. *Component Software – Beyond Object-Oriented Programming – Second Edition*. ACM Press. Addison-Wesley, New York, NY, 2002.
[SGVVRF04]	Eduardo Souto, Germano Guimarães, Glauco Vasconcelos, Mardoqueu Vieira, Nelson Rosa, Carlos Ferraz. *A Message-Oriented Middleware for Sensor Networks*. In proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing, Toronto, Ontario, Canada, October 18-22, 2004.
[SH05]	Séverine Sentilles, Natacha Hoang. *Exploration du modèle de composant Fractal*. Rapport de projet tutoré, Pau, 2005.
[Th06]	*Think Home Page.* Last modified at 2006-03-31 http://think.objectweb.org/
[TOS04]	*TinyOS Mission Statement*. UC Berkeley, 2004. http://www.tinyos.net/special/mission
[TOS05]	*TinyOS/nesC Overview*. In a Smart Dust Training Seminar CD, San Jose, February 9-10, 2005.
[YB05]	E. Yoneki and J. Bacon. *A Survey of Wireless Sensor Network Technologies: Research Trends and Middleware's Role.* Technical Report UCAM-CL-TR646, University of Cambridge, 2005.
[YKP04]	Yang Yu, Bhaskar Krishnamachari, and Viktor K. Prasanna. *Issues in designing middleware for wireless sensor networks*. In IEEE Network, Vol. 18, No.1, pp. 15-21, 2004.
[Wei96]	Mark Weiser, *Ubiquitous Computing*. 03/17/96 http://www.ubiq.com/hypertext/weiser/UbiHome.html
[Wik06a]	Wikipédia – Atmel AVR. 2006. http://en.wikipedia.org/wiki/Atmel_AVR
[Wik06b]	Wikipédia - ARM architecture. 2006. http://en.wikipedia.org/wiki/ARM_architecture

Table des figures

FIGURE 1 : SCHEMATISATION D'UN CAPTEUR "TRADITIONNEL" ---- 4 -
FIGURE 2 : SCHEMA D'UN COMPOSANT D'UN RESEAU DE CAPTEURS, INSPIRE DE [KDM05] ---- 6 -
FIGURE 3 : MODELE DE CAPTEUR VIRTUEL, INSPIRE DE [LEW04] ---- 7 -
FIGURE 4 : SCHEMATISATION D'UN RESEAU DE CAPTEURS SANS-FIL ---- 7 -
FIGURE 5 : SCHEMATISATION D'UN COMPOSANT TINYOS [HSWHCP00B] ---- - 12 -
FIGURE 6 : GRAPHE DE COMPOSANTS D'UNE APPLICATION VISANT AU ROUTAGE DES RELEVES DES CAPTEURS [HSWHCP00A] ---- - 13 -
FIGURE 7 : MODELISATION DE L'UTILISATION DE LA MEMOIRE D'UN MOTE [TOS05] ---- - 13 -
FIGURE 8 : MODELISATION DE L'ORDONNANCEUR TINYOS [LIU05] ---- - 14 -
FIGURE 9 : LES ETAPES DE LA COMPILATION D'UNE APPLICATION TINYOS[MAU] ---- - 17 -
FIGURE 10 : ILLUSTRATION DE LA NOTION DE COMPOSANT ---- - 18 -
FIGURE 11 : REPRESENTATION UML DES 2 TYPES D'INTERFACES EXISTANTES ---- - 19 -
FIGURE 12 : SCHEMA UML 2.0 D'UNE LIAISON ---- - 19 -
FIGURE 13 : REPRESENTATION D'UNE INTERFACE A L'EXECUTION [FSLM02] ---- - 21 -
FIGURE 14 : REPRESENTATION DE L'INTERFACE UNIQUE D'UN COMPOSANT [FSLM02] ---- - 21 -
FIGURE 15 : REPRESENTATION D'UNE INTERFACE DANS LE CAS GENERAL [FSLM02] ---- - 21 -
FIGURE 16 : SYSTEME A BASE DE COMPOSANTS, DE LIAISONS ET DE DOMAINES [FAS01] ---- - 22 -
FIGURE 17 : LA CHAINE DE CONSTRUCTION DE THINK [LOB] ---- - 23 -
FIGURE 18 : VUE INTERNE D'UN COMPOSANT ---- - 24 -
FIGURE 19 : VUE EXTERNE D'UN COMPOSANT [BRU03] ---- - 25 -
FIGURE 20 : "COMPONENTIZATION" D'UN CAPTEUR ---- - 29 -
FIGURE 21 : LIAISON ENTRE LE SYSTEME D'EXPLOITATION ET L'APPLICATION ---- - 30 -
FIGURE 22 : LES RESSOURCES SUR UN CAPTEUR ---- - 31 -
FIGURE 23 : LES ETATS D'UN CAPTEUR ---- - 32 -
FIGURE 24 : MODELISATION DE L'UTILISATION D'UNE FILE D'EVENEMENT ---- - 33 -
FIGURE 25 : MODELE DE COMPOSANT BASE EVENEMENT ---- - 33 -
FIGURE 26 : FONCTIONNEMENT DU SYSTEME ---- - 34 -
FIGURE 27 : LIAISON ENTRE LE CONTROLEUR DE RECONFIGURATION ET LE SYSTEME A RECONFIGURER [P04] - - 36 -
FIGURE 28 : IMPLEMENTATION DES *PROXIS* D'UN COMPOSANT [POL04] ---- - 37 -
FIGURE 29 : LES ETATS DU SYSTEME ---- - 38 -

Annexes

ANNEXE A : LES COMPOSANTS SYSTEMES FOURNIS POUR POWERPC DANS LA BIBLIOTHEQUE KORTEX..... - 48 -
ANNEXE B : MANUEL D'INSTALLATION DE THINK-V2 POUR UBUNTU ..... - 49 -
ANNEXE C : MANUEL D'INSTALLATION DE THINK-V3 POUR UBUNTU ..... - 52 -
ANNEXE D : MANUEL D'INSTALLATION ET D'UTILISATION DE L'EMULATEUR ARM SKYEYE-V1 (POUR UBUNTU) ..... - 55 -
ANNEXE E : DEVELOPPEMENT D'UNE APPLICATION DE TEST ..... - 56 -
ANNEXE F : MANUEL DE L'IDL ..... - 60 -

Annexe A :
Les composants systèmes fournis pour PowerPC dans la bibliothèque Kortex

Interface	Attribut	Composant	Description	Lignes
		Mémoire		
MMU		mmu	MMU PowerPC	280
Page	"page"	buddy	buddy-system	270
Allocator	"allocator"	dlmalloc	allocateur	474
Sbrk	"sbrk"	sbrkmap	sbrk mémoire paginée	94
"	"	sbrk	sbrk mémoire plate	72
Space	"space"	flat	mémoire plate	134
"	"	pagetable	mémoire paginée	292
SpaceFactory	"spacefactory"	spacefactory	usine à espaces d'adressage	65
		Processeur		
Trap		trap	exceptions PowerPC	109+90[a]
Scheduler	"Scheduler"	roundrobin	ordon. préemptif circulaire	396
"	"	priority	ordon. préemptif à priorités	415
"	"	cooperative	ordon. coopératif	377
irq	"irq"	IRQ	gestionnaire interruption	121
		Domaine		
Trader		trader	mini-courtier	91
Loader	"loader"	loader	chargeur dynamique	341
Starter	"starter"	starter	chargeur application	310
		Réseau		
net	"eth"	mace	pilote carte mace	659
"	"	bmac	pilote carte bmac	855
"	"	gmac	pilote carte gmac	1063
"	"	tulip	pilote carte PCI tulip	2208
PacketFactory	"packetfactory"	packet	usine à paquets	149
ProtocolHigh	"ethernet"	ethernet	protocole Ethernet	417
"	"iphigh"	ip	protocole IP	476
"	"udphigh"	udp	protocole UDP	279
"	"tcphigh"	tcp	protocole TCP	802
"	"rpchigh"	rcp	protocole RPC	335
Arp	"arp"	arp	protocole ARP	240
Socket	"tcp"	tcp	socket TCP	
"	"ucp"	ucp	socket UDP	
VFS	"nfs"	nfs	protocole NFS	752
		Graphique		
framebuffer	"fb"	offb	pilote carte vidéo	366
font	"font"	F8x16	police de caractères	4667
console	"console"	console	console	129
		Contrôleurs		
FW	"FW"	firmware	pilote firmware	821
pci	"pci"	uninorth	pilote PCI uninorth	315
"	"	grackle	pilote PCI grackle	243
"	"	pmac	pilote PCI bandit & chaos	398
pic	"pic"	openpic	pilote PIC openpic	405
"	"	pmacpic	pilote PIC macio & GC	198
via	"via"	pmu	pilote ADB pmu	692
"	"	cuda	pilote ADB cuda	450

Annexe B :
Manuel d'installation de Think-v2 pour Ubuntu

Pré requis :

- Nécessite d'avoir cvs installé
- Un environnement de développement C (avec gcc, as, ar, ld)
- La librairie libc6-dev (permet d'avoir le fichier crt1.o)

Logiciel à installer :

- Think-v2
- JDK 1.4.2
- Ant 1.6.5
- Un compilateur croisé pour arm

Procédure d'installation :

1. Télécharger les sources Think de la manière suivante :
   Taper la commande suivante :
   - cvs -z3 -d :pserver :anonymous@cvs.forge.objectweb.org/cvsroot/think co .
     («.» signifie que la version téléchargée sera installée dans le répertoire courant.)

2. Installer l'environnement de développement
   2.1. Installer le JDK 1.4.
   Pour des raisons de compatibilités avec un autre logiciel à installer sur notre machine de développement, nous avons choisi d'installer le JDK fournit par IBM et non celui fournit par SUN. Normalement, la démarche pour installer celui de SUN est similaire.
   a. Télécharger l'archive IBMJava2-SDK-142.tgz sur le site d'IBM
      (Nécessite la création d'un compte)
   b. Passer en mode root :
      - su + password
   c. Décompresser l'archive dans le répertoire /opt
      - tar -zxvf IBMJava2-SDK-142.tgz -C /opt
   d. « Suppression » du java préexistant
      (installé par l'Ubuntu par défaut)
      - mv /usr/bin/java /usr/bin/java_initial
   e. Modifier la variable d'environnement PATH et ajout des variables d'environnement adéquates dans le .bash_profile de l'utilisateur
      - exit
      - gedit ~/.bash_profile &

      ```
      PATH=/opt/IBMJava2-142/bin:/opt/IBMJava2-142/jre/bin :$PATH
      JAVA_HOME=/opt/IBMJava2-142
      export JAVA_HOME
      export PATH
      ```
   2.2. Installer ant

a. Télécharger l'archive apache-ant-1.6.5.bin.tar.gz sur le site d'apache
b. Passer en mode root :
   - su + password
c. Décompresser l'archive dans le répertoire /opt
   - tar -zxvf apache-ant-1.6.5.bin.tar.gz -C /opt
d. Modifier la variable d'environnement PATH et ajout de la variable d'environnement adéquate dans le .bash_profile de l'utilisateur
   - exit
   - gedit ~/.bash_profile &

```
PATH=/opt/IBMJava2-142/bin:/opt/IBMJava2-142/jre/bin:
        /opt/apache-ant-1.6.5/bin :$PATH
ANT_HOME=/opt/apache-ant-1.6.5
export ANT_HOME
```

2.3. Installer le compilateur croisé pour arm

Ceci est la partie la plus délicate car il faut trouver un compilateur compatible avec la machine de développement, capable de compiler en C et en assembleur et de faire des éditions de liens.

En cherchant « cross-3.2 » sur google, nous en avons trouvé un pour hôte : linux/x86 et pour cible : familiar/arm.

Ce manuel décrit la procédure suivie pour cette installation

a. Décompresser l'archive dans le répertoire /usr/local
   - tar -zxvf cross-3.2.tar.gz -C /
b. Modifier la variable d'environnement PATH et ajout de la variable d'environnement adéquate dans le .bash_profile de l'utilisateur
   - gedit ~/.bash_profile &

```
CROSS_COMP=/usr/local/arm/bin
PATH=$CROSS_COMP :/opt/IBMJava2-142/bin:/opt/IBMJava2-
        142/jre/bin:/opt/apache-ant-1.6.5/bin :$PATH
```

   - source ~/.bash_profile

3. Compiler les outils de Think, puis les bibliothèques nécessaires
   - cd ~/v2
   - ant tools

     Si la compilation génère une erreur de type MalformedInputException, il est nécessaire d'exécuter la commande suivante : export LANG=en_US et puis de relancer la commande ant tools. Cette erreur est apparemment liée à l'installation de JDK d'IBM. Elle ne semble pas se produire si l'on utilise celui de Sun.
   - ant -DOS=Think -DCPU=arm
   - ant -DOS=Linux -DCPU=arm
   - ant -DOS=Unix
4. Compiler un des exemples
   a. Pour une cible unix :
      - cd ~/v2/test/helloworld
      - ant unix

        Si le message d'erreur suivant se produit « Unix ${CPU} could not be determine », deux solutions sont envisageables :

- soit modifier le fichier $THINKPATH/tools/org/objectweb/think/helper/Think.java en rajoutant :

```
…
else if(« i386 ».equals(osarch) || « x86 ».equals(osarch) )
```

```
CPU = new String(« x86 ») ;
…
```

puis recompiler les outils et les librairies dans le répertoire $THINKPATH avant de relancer la commande ant unix

- soit ajouter la ligne suivante dans le fichier : $THINKPATH/test/helloworld/build.xml

```
<target name=« unix »>
…
<property name=« OS » value=« unix »/>
<property name=« CPU » value=« x86 »/>
…
</target>
```

et relancer la commande ant unix.

b. Pour une cible arm :
   - cd ~/v2/example/helloworld
   - ant arm

     Si une erreur se produit au [ld] de type référence indéfinie vers « memcpy_P ». Il faut copier le fichier $THINKPATH/src/libc/powerpc/memcpy_P.c dans $THINKPATH/src/libc/ et relancer la compilation.

Il est à noter qu'à l'heure où nous rédigeons ce manuel, la version 2 est la dernière release.

Annexe C :
Manuel d'installation de Think-v3 pour Ubuntu

Pré requis :

- Nécessite d'avoir svn installé
- Un environnement de développement C (avec gcc, as, ar, ld)
- La librairie libc6-dev (permet d'avoir le fichier crt1.o)

Logiciel à installer :

- Think-v3
- JDK 1.4.2
- Ant 1.6.5
- Un compilateur croisé pour arm

Procédure d'installation :

5. Télécharger les sources Think de la manière suivante :
   a. Passer en mode root :
      - su + password
   b. Taper la commande suivante :
      - svn checkout svn://svn.forge.objectweb.org/svnroot/think v3
        (v3 représente le répertoire dans lequel seront téléchargées les sources)
   c. Changer les droits d'utilisation :
      - chown nomUtilisateur v3 -R
   d. Ajouter les variables d'environnement THINKOWPATH et THINKPATH dans le .bash_profile de l'utilisateur
      - gedit ~/.bash_profile &

      THINKPATH = ~/v3/trunk
      THINKOWPATH = ~/v3/trunk
      export THINKPATH
      export THINKOWPATH

6. Installer l'environnement de développement

   2.4. Installer le JDK 1.4.

   Pour des raisons de compatibilités avec un autre logiciel a installé sur notre machine de développement, nous avons choisi d'installer le JDK fournit par IBM et non celui fournit par SUN. Normalement, la démarche pour installer celui de SUN est similaire.

   a. Télécharger l'archive IBMJava2-SDK-142.tgz sur le site d'IBM
      (Nécessite la création d'un compte)
   b. Passer en mode root :
      - su + password
   c. Décompresser l'archive dans le répertoire /opt
      - tar -zxvf IBMJava2-SDK-142.tgz -C /opt
   d. « Suppression » du java préexistant
      (installé par l'Ubuntu par défaut)
      - mv /usr/bin/java /usr/bin/java_initial

e. Modifier la variable d'environnement PATH et ajout des variables d'environnement adéquates dans le .bash_profile de l'utisateur
   - exit
   - gedit ~/.bash_profile &

```
PATH=/opt/IBMJava2-142/bin:/opt/IBMJava2-142/jre/bin :$PATH
JAVA_HOME=/opt/IBMJava2-142
export JAVA_HOME
export PATH
```

2.5. Installer ant
   a. Télécharger l'archive apache-ant-1.6.5.bin.tar.gz sur le site d'apache
   b. Passer en mode root :
      - su + password
   c. Décompresser l'archive dans le répertoire /opt
      - tar -zxvf apache-ant-1.6.5.bin.tar.gz -C /opt
   d. Modifier la variable d'environnement PATH et ajout de la variable d'environnement adéquate dans le .bash_profile de l'utilisateur
      - exit
      - gedit ~/.bash_profile &

```
PATH=/opt/IBMJava2-142/bin:/opt/IBMJava2-142/jre/bin:
        /opt/apache-ant-1.6.5/bin :$PATH
ANT_HOME=/opt/apache-ant-1.6.5
export ANT_HOME
```

2.6. Installer le compilateur croisé pour arm

Ceci est la partie la plus délicate car il faut trouver un compilateur compatible avec la machine de développement, capable de compiler en C et en assembleur et de faire des éditions de liens.

En cherchant « cross-3.2 » sur google, nous en avons trouvé un pour hôte : linux/x86 et pour cible : familiar/arm.

Ce manuel décrit la procédure suivie pour cette installation

   a. Décompresser l'archive dans le répertoire /usr/local
      - tar -zxvf cross-3.2.tar.gz -C /
   b. Modifier la variable d'environnement PATH et ajout de la variable d'environnement adéquate dans le .bash_profile de l'utilisateur
      - gedit ~/.bash_profile &

```
CROSS_COMP=/usr/local/arm/bin
PATH=$CROSS_COMP :/opt/IBMJava2-142/bin:/opt/IBMJava2-
        142/jre/bin:/opt/apache-ant-1.6.5/bin :$PATH
```

      - source ~/.bash_profile

7. Compiler les outils de Think, puis les bibliothèques nécessaires
   - cd ~/v3
   - ant tools

     Si la compilation génère une erreur de type MalformedInputException, il est nécessaire d'exécuter la commande suivante : export LANG=en_US et puis de relancer la commande ant tools. Cette erreur est apparemment liée à l'installation de JDK d'IBM. Elle ne semble pas se produire si l'on utilise celui de Sun.
   - ant -DOS=Think -DCPU=arm
   - ant -DOS=Linux -DCPU=arm
   - ant -DOS=Unix
8. Compiler un des exemples
   a. Pour une cible unix :

- cd ~/v3/example/helloworld
- ant unix

  Si le message d'erreur suivant se produit « Unix ${CPU} could not be determine », deux solutions sont envisageables :

- soit modifier le fichier $THINKPATH/src/helper/Think.java en rajoutant :

```
…
else if(« i386 ».equals(osarch) || « x86 ».equals(osarch) )
CPU = new String(« x86 ») ;
…
```

puis recompiler les outils et les librairies dans le répertoire $THINKPATH avant de relancer la commande ant unix

- soit ajouter la ligne suivante dans le fichier :
  $THINKPATH/example/helloworld/build.xml

```
<target name=« unix »>
…
<property name=« OS » value=« unix »/>
<property name=« CPU » value=« x86 »/>
…
</target>
```

et relancer la commande ant unix.

c. Pour une cible arm :
   - cd ~/v3/example/helloworld
   - ant arm

     Si une erreur se produit au [ld] de type référence indéfinie vers « memcpy_P ». Il faut copier le fichier $THINKPATH/src/generic/libc/powerpc/memcpy_P.c dans $THINKPATH/src/generic/libc/ et relancer la compilation.

Il est à noter qu'à l'heure où nous rédigeons ce manuel, la version 3 est en cours de développement. Elle est donc continuellement sujette à modification et tous les exemples ne compilent pas. Pour plus d'information, consulter les liens « Bugs » et « Listes de diffusion » présents à l'adresse suivante : http://forge.objectweb.org/ projects/think/

Annexe D :
Manuel d'installation et d'utilisation de l'émulateur ARM SkyEye-v1 (pour Ubuntu)

Procédure d'installation :

- Vérifier dans Synaptic que les paquets suivants sont bien installés. Dans le cas contraire les installer.
  - libgtk2.0-dev
  - pkg-config
  - libatk1.2-dev
  - libpango1.0-dev
  - freetype2.0-dev
  - libglib2.0-dev
  - libx11-dev
  - binutils-dev
- créer un lien symbolique du gcc existant vers un gcc-3.3
  - su
    (password)
  - ln -sf /usr/bin/gcc /usr/bin/gcc-3.3
- télécharger l'archive skyeye-1.2-RC7-2.tar.bz2 à l'adresse http://gro.clinux.org/frs/2group_id=327 et l'extraire dans un répertoire (comme par exemple ~/ARMEMU)
  - cd ~/ARMEMU/skyeye-v1
  - make
    l'exécutable se trouve alors dans le répertoire ~/ARMEMU/skyeye-v1/binary
  - cd binary
  - ./skyeye -h pour afficher une liste des options disponibles.

Procédure pour tester l'installation et pour comprendre comment configurer skyeye

- Télécharger l'archive testsuite2.1.ter.bz2 et la décompresser dans un répertoire tel que ~/ARMEMU/
- Tester les exemples fournis
- Ouvrir un des fichiers skyeye.conf. Celui-ci contient un découpage de la mémoire en section. Au cours de nos tests, nous avons utilisé le fichier skyeye.conf pour sa1100 qui correspond à un processeur ARM.

Commande pour utiliser pour lancer l'émulateur

- ./ARMEMU/skyeye-v1/binary/skyeye -e emplacementImageKernel -c emplacementFichierSkyeye.conf

Il est à noter qu'avec ce fichier de configuration seuls les exemples Think pour les noyaux ARM 3600, 3800 fonctionnent. Pour les noyaux 2200, 3900 et 5400 une erreur d'écriture en mémoire se produit : « sa_io_write_byte error ».

Annexe E :
Développement d'une application de test

But :

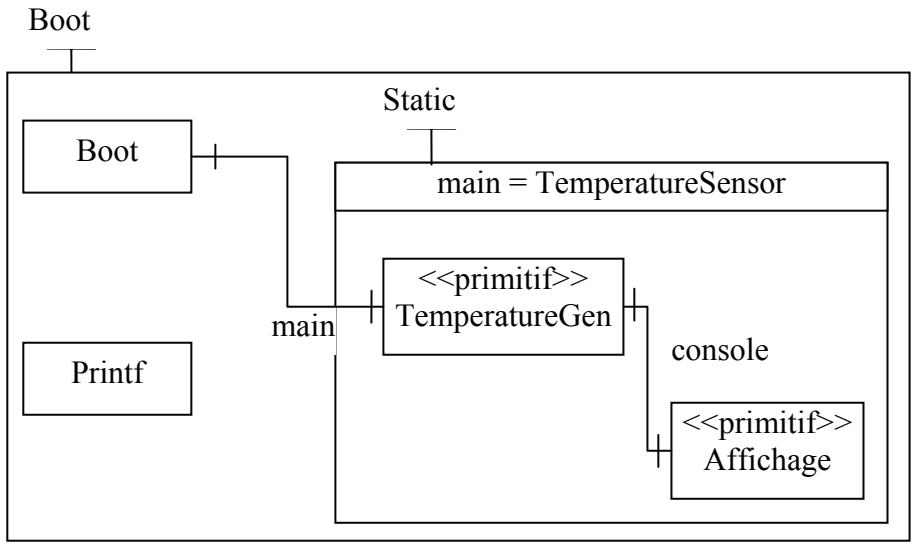
Comprendre comment créer des composants et des applications Think.

Application envisagée :

1 générateur de température aléatoire. Ce dernier aurait pu en cas de succès du développement de l'application permettre de simuler un capteur.

Version 1 :

Adaptation de l'exemple helloworld fourni. Voici la modélisation de l'application envisagée.



Des exemples de fichiers ADL :

Nous présentons d'abord le fichier `tpkernel.adl` qui permet de décrire l'architecture globale de l'application. Celle-ci est constituée des composants `boot`, `main` et `printf`. Le composant `boot` est implémenté par le composant `arm.sa1100.boot.lib.boot`, le composant `main` par `temperatureSensor` et `printf` par `log.lib.putcprint`. Cela est mis en évidence par le mot-clé `contains`. De plus, main possède un sous-composant `console` qui est réimplémenté par le composant `affichage`. Cela est mis en évidence par `overloads`. Enfin, le contrôleur est indiqué par le mot-clé `controller`.

```
composite tpkernel implements RootType {
    contains boot = arm.sa1100.boot.lib.boot
    contains main = temperatureSensor
    overloads main/console = affichage
     //overloads main/console = ipaq.h3600.video.lib.screen
     /* this must be remove when printf will be manage well*/
    contains printf = log.lib.putcprint
    controller org.objectweb.think.controller.Boot
}
```

Voici le fichier d'un composant composite : `temperatureSensor.adl`. Comme on peut le constater sur cette description grâce aux mots-clés `binds` et `to`, l'interface `main` requise par ce composant est liée à l'interface fournie par le composant `tp`.

```
composite temperatureSensor implements RootType {
      contains console : ServerType
      contains tp = temperatureGen
      binds this.main to tp.main
      binds tp.console to console.console
      controller org.objectweb.think.controller.Static
}
```

Enfin voici la description d'un composant primitif : `temperatureGen.adl`. Celui-ci fournit et requiert les interfaces mentionnées dans le fichier `ClientType.adl`. De plus, le composant est généré automatiquement par les outils Think en se basant sur le code fonctionnel contenu dans `temperatureGen.c`. Enfin, tous les composants `temperatureGen` créés seront strictement identiques (interfaces fournies et requises, attributs utilisés, etc.). Cela est mentionné via le mot-clé `template`.

```
primitive temperatureGen implements ClientType {
      skeleton temperatureGen nolifecycle
      template
    }
```

Un exemple de fichier contenant le code fonctionnel d'un composant primitif :

```
#include <activity/api/Main.idl.h>
#include <video/api/Console.idl.h>


struct temperatureGendata {
      // Imported interfaces
      Rvideo_api_Console      *console;
};

#if ! defined(ONLYDEFINITION)

#include <kortex.h>

/*
 * Template interface
 */
int randomize_temperature(jint max) {
return (max-1);
}

static void mainentry(void* _this, jint argc, char** argv) {
  DECLARE_SELF(struct temperatureGendata);
  if (self->console)
      {
            CALL3(self->console, putxycs, 15, 15, "TEST");
      int cpt=0;
      int temp=19;
```

```
            while(cpt<10)
                  {
                  temp= randomize_temperature(temp);
                  printf("temperature = '%d'\n", temp);
                  cpt++;
                  }
            }

        /*while(1);*/
      }

      struct Mactivity_api_Main temperatureGen_mainmeth = {
        main: mainentry,
      };

      #endif
```

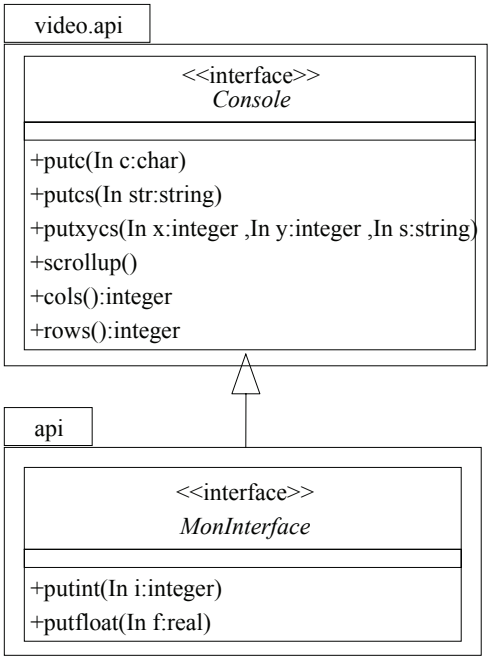
Conclusion :

- Le composant `Printf` s'avère nécessaire pour les affichages sur la machine de cible car la fonction `printf()` pour arm ne fonctionne pas correctement
- `Boot` et `Static` sont les deux contrôleurs de la version initiale. Cependant la présence de `Boot` est indispensable pour charger l'exécutable
- Pour que la compilation s'exécute correctement, nous avons récupéré le fichier le fichier `hw_sa.lds` présent dans le répertoire example/helloworld/arm. Celui-ci définit les sections mémoires utilisées lors de l'édition de liens. Voici comment il se présente :

```
      SECTIONS
      {
          . = 0xC0008000;
          _stext = .;
          .text : { *(.start) *(.text*) *(.gnu.linkonce.t.*)}
          .rodata : { *(.rodata*) *(.gnu.linkonce.r.*) *(.init) *(.glue_7)
*(.glue_7t)}
          .data : { *(.data*) *(.gnu.linkonce.d.*)}
           _edata = .;

            . += 4096;
          kernelstackend = .;
            . += 4096;
          trapstackend = .;

             __bss_start = .;
         .bss : { *(.bss*) *(.gnu.linkonce.b.*) }
            __bss_end = .;

          heapstart = .;
          _end = .;
      }
```

Version 2 :

Modification du composant affichage pour afficher des entiers à l'écran. (Test de l'héritage entre interfaces).



Réalisation :

- Création d'un fichier api/MonAffichage.idl contenant la déclaration des fonctions putint et putfloat.
- Modification des fichiers adéquats (ClientType, Server Type) pour définir MonAffichage comme l'interface fournie et requise dans affichage.adl.
- Modification du fichier temperatureGen.c pour modifier les liens vers les interfaces (notamment modification des en-têtes des librairies et du pointeur vers l'interface importée).
- Modification du fichier affichage.c pour :
    - Modifier l'en-tête
    - Définir les fonctions rajoutées
    - Modifier la structure MonInterface

Conclusion :

- Au niveau des interfaces :

Bien que la possibilité de pouvoir faire de l'héritage entre interface soit indiquée dans le tutoriel de think version 3, cette fonctionnalité n'est pas encore implémentée. En effet, seules les fonctions déclarées dans l'interface fille sont connues. Pour pallier à ce problème, nous avons dû redéclarer toutes les fonctions contenues dans l'interface mère dans l'interface fille.

- Au niveau de la fonction printf :

A l'heure où nous rédigeons ce rapport, le %f génère toujours une erreur. Nous pensons que ce problème provient d'une mauvaise installation du compilateur croisé pour arm. En effet, les messages obtenus à la compilation, nous indique un conflit entre l'utilisation de réels programmés en hardware et l'utilisation de réels programmés en software. Il semble être nécessaire de réaliser l'installation en spécifiant l'une, l'autre ou même les deux options suivantes : -msoftfloat, -without-FP. Malheureusement, cela requiert de trouver les versions de gcc, glibc et arm compatibles entre elles ainsi qu'avec ces options. Nous avons réalisé plusieurs tentatives à l'aide d'un script mais nous avons toujours abouti à un échec.

Annexe F :
Manuel de l'IDL

Think IDL

In order to allow Fractal components implemented in potentially distinct programming languages to interoperate, some standard protocols for local and remote operation invocations are necessary. One way to ensure this is to use an Interface Definition Language (IDL), and mappings from the IDL to existing programming languages. The IDL compiler can then generate stubs and skeletons to make the conversion from language specific protocols to standard protocols, and vice versa.

The IDL language used in Think is a modified subset of Java, so as to be immediately understandable by Java programmers.

Grammar

There are two forms or identifiers: simple identifier and qualified identifiers. One limitation here is to not employed C reserved keywords in identifier.

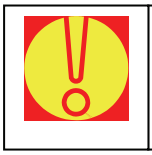
```
QualifiedIndentifer ::= ( Identifier ( "."Identifier )*
```

An interface declaration is a package declaration followed by the interface type declaration. The file containing the interface must reside in the directory designate by the package QualifiedIdentifer.

```
Declaration ::= "package" QualifiedIdentifer ";" InterfaceDeclaration
```

An interface type declaration identified by Identifier contains two sorts of declarations; methods and constant fields. The keyword public isn't relevant and is only here for compatibility purpose. The fully qualified name of an interface is the package QualifiedIdentifier plus the interface Identifer separate by a point. A interface could inherit from one single interface. You can declare as many as require interleaved fields and methods in an interface declaration.

```
InterfaceDeclaration ::=  "public"? "interface" Identifier "extends"
QualifiedIdentifer "{"
(
 MethodDeclaration |
 FieldDeclaration |
 ";")*
"}"
```



A l'heure où nous rédigeons ce rapport l'héritage d'interface n'est pas implémenté

A method declaration grammar looks like this. The keyword ... is here for passing arbitrary parameters number.

```
MethodDeclaration ::= ResultType Identifier FormalParameters ;"
ResultType ::= Type | "void"
FormalParameters ::=
 "("
 FormalParameter
 ( "," FormalParameter )*
 ( "," "..." ) ?
 ")"
FormalParameter ::= "const"? Type Identifier
```

A constant field declaration grammar looks like this. The expression must be a valid constant expression and only primitive type can be constant. Actually, no semantic check on expression was done, and the expression was directly injected in generated C code.

```
FieldDeclaration ::= PrimitiveType Identifier = Expression ";"
```

The IDL understand these types. Where any means that all interface reference types is accepted and ComplexeType specifically designs an interface reference type.

```
Type ::=
  (PrimitiveType | ComplexeType) ( "*" | ("[" "]") )*
ComplexeType ::= QualifiedIndentifer | "any"
PrimitiveType ::= (
"boolean" |
"char" |
"byte" | "unsigned" "byte" |
"short" | "unsigned" "short" |
"int" | "unsigned" "int" |
"long" | "unsigned" "long" |
"float" |
"double" |
"string")
```